НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Теплоенергетичний факультет
Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено
Завідувач кафедри
_____ О.В. Коваль
(підпис) (ініціали, прізвище)
“___”_____2019 р.

ДИПЛОМНА РОБОТА

на здобуття ступеня бакалавра

з напряму підготовки 6.050103 “ Програмна інженерія “

на тему: Система процедурної генерації та уніфікації анімацій на основі інверсної кінематики

Виконала: студент 4_курсу, групи__ТВз-51__

_____Шинкар Дмитро Олександрович_____ _____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____к.е.н., Гусєва І.І._____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент директор ТОВ «Юбісофт-Юкрейн»_Щербанеску А.Г. _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2019

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший , бакалаврський

Напрям підготовки 6.050103 “ Програмна інженерія”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.В. Коваль
(підпис)
”___”_____2018р.

ЗАВДАННЯ

на дипломну роботу студенту

_____Шинкарю Дмитру Олександровичу_____
(прізвище, ім’я, по батькові)

1.Тема роботи Система процедурної генерації та уніфікації анімацій на основі інверсної кінематики
керівник роботи Гусєва Ірина Ігорівна, к.е.н.
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від ”__”___ 2018р. № __

2. Строк подання студентом роботи _____

3. Вихідні дані до роботи: Система процедурної генерації та уніфікації анімацій на основі інверсної кінематики

4.Зміст розрахунково-пояснювальної записки (перелік завдань, які потрібно розробити) розробити проміжний формат анімацій на основі інверсного скелету, пронаналізувати і реалізувати алгоритми інверсної кінематики і елементи процедурної поведінки, імплементувати експорт анімації у традиційний формат.

5. Перелік ілюстративного матеріалу: схеми архітектури додатку, знімки екранних форм, діаграма прецедентів системи, діаграма класів, діаграма структури системи, зразки розробленого інтерфейсу додатку.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-6	Шалденко О.В. к.т.н., ст.викл.		

7. Дата видачі завдання ”__”_листопада_____2018 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Вивчення та аналіз задачі		
2	Розробка архітектури та загальної структури системи		
3.	Розробка структур окремих підсистем		
4.	Програмна реалізація системи		
5.	Оформлення пояснювальної записки		
6.	1 Захист програмного продукту		
7.	2 Передзахист		
8.	Захист		

Студент _____ Шинкар Д. О.

(підпис) (прізвище та ініціали,)

Керівник роботи _____ Гусєва І. І.

(підпис) (прізвище та ініціали,)

АНОТАЦІЯ

Метою роботи було створення універсального проміжного формату скелетної анімації оснований на ієрархії ланцюжків кісток і методу перетворення традиційних скелетних анімацій у цей проміжний формат, а також аналіз існуючих алгоритмів інверсної кінематики для реалізації програвання проміжного формату і можливості запису анімацій у традиційному форматі.

Створена програма здатна перетворювати анімацію в проміжний формат, переносити її на іншого персонажа або додавати процедурну поведінку і записувати знову у традиційному форматі, забезпечуючи зручний інтерфейс користувача і можливість перегляду анімації на різних персонажах на усіх етапах її перетворення.

Для забезпечення переносу було реалізовано кілька сучасних алгоритмів вирішення задачі інверсної кінематики. Кінцевий результат роботи у програмі являє собою анімацію у традиційному форматі, перенесену на персонажа із скелетом, що має структурні відмінності відносно скелету персонажа, для якого ця анімація створювалась.

Ключові слова: скелетна анімація, пряма кінематика, інверсна кінематика, перенаправлення анімацій, процедурна анімація.

Обсяг звіту становить 52 сторінки, міститься 21 ілюстрація, 3 додатки. Загалом опрацьовано 15 джерел.

ABSTRACT

The aim of the work was to create a universal intermediate skeleton animation format based on the bone chain hierarchy and the method of converting traditional skeletal animations into this intermediate format, as well as an analysis of existing inverted kinematics algorithms for implementing intermediate format playback and the ability to record animations to a traditional format.

The created program is capable of converting an animation into an intermediate format, transferring it to another character, or adding procedural behavior and recording it again as a traditional animation, providing a user-friendly interface and the ability to view animations on different characters at all stages of its transformation.

Several modern algorithms for solving the inverse kinematics problem were implemented to ensure the transfer of animations. The end result of the program is an animation in the traditional format, transferred to a character with a skeleton that has structural differences with respect to the skeleton of the character for which this animation was created.

Key words: skeleton animation, forward kinematics, inverse kinematics, retargeting of animations, procedural animation.

The volume of the report is 52 pages, it contains 21 illustrations and 3 attachments. In total, 15 sources have been processed.

ЗМІСТ

Перелік умовних позначень, скорочень і термінів 8
Вступ 9
1. ПОСТАНОВКА ЗАДАЧІ СТВОРЕННЯ СИСТЕМИ ГЕНЕРАЦІЇ ТА УНІФІКАЦІЇ АНІМАЦІЙ 11
2. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ УНІФІКАЦІЇ АНМАЦІЙ 14
2.1. Аналіз предметної області: створення скелету 14
2.2. Аналіз існуючих програмних засобів для уніфікації анімацій 16
2.3. Висновки 17
3. АНАЛІЗ АЛГОРИТМІВ ВИРІШЕННЯ ЗАДАЧІ ІНВЕРСНОЇ КІНЕМАТИКИ 18
3.1 Детальний опис проблеми інверсної кінематики 18
3.2 Алгоритми на основі Якобіана 19
3.3 Ньютонівські методи 22
3.4 Послідовний метод Монте-Карло 23
3.5 Циклічно-координатний спуск 23
3.6 FABRIK 25
3.6.1 Опис алгоритму 26
3.6.2 Застосування для декількох ефекторів 28
3.6.3 Застосування із обмеженнями поворотів 30
3.7 Висновки 32
4. ЗАСОБИ РЕАЛІЗАЦІЇ ПРОГРАМНОЇ СИСТЕМИ 34
4.1. Опис архітектури застосунку 34
4.2 Опис інструментів розробки 36
4.3 Обґрунтування вибору програмної реалізації 38
4.4 Висновки 39
5. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ 40
5.1 Опис функціональності системи 40
5.2 Метод завантаження і програшу анімацій 41
5.3 Проміжний формат анімацій 43
5.4 Програвання проміжного формату і експорт 45

6. МЕТОДИКА РОБОТИ КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ 46
6.1 Інсталяція та системні вимоги 46
6.2 Інструкція з використання програмного продукту 46
ВИСНОВКИ 51
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 52
ДОДАТОК А 54
ДОДАТОК Б 56
ДОДАТОК В 73

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Анімація – процес зміни деякого параметру об’єкту у часі. У контексті скелетної анімації – процес зміни орієнтації, розміру і положення кісток віртуального скелету.

Віртуальний скелет – ієрархічна структура, що складається із кісток, що представлені орієнтацією, розміром і позицією, зазвичай використовується для керування сіткою(мешем) моделі персонажу.

Рушій – система візуалізації тривимірних сцен.

DoF – ступінь свободи (degree of freedom) це кількість осей, по яким одночасно можуть рухатися або обертатися прикріплені частини, в тому числі кістки віртуального скелету.

ВСТУП

У наш час системи скелетної анімації мають надзвичайно широкі сфери застосунку. Вони використовуються як у робототехніці для маніпулювання руками роботів що здійснюють операції для виконання різноманітних технологічних процесів на підприємствах так і у індустрії анімаційних фільмів, де віртуальні скелети керують рухами персонажів, забезпечуючи їх художню виразність і характер. Не варто також забувати про величезну індустрію відеоігор, де системи скелетної анімації використовуються надзвичайно широко, але на відміну від анімаційних фільмів у іграх вони працюють у реальному часі.

Існує чимало інструментів створення і конфігурації скелетних анімацій, які використовуються у різних сферах в залежності від своєї специфікації. Вони дають широкий вибір методів створення анімацій але все ж процес створення анімацій буває в основному двох видів: запис рухів за допомогою різноманітних інструментів і систем motion capture або ж самостійне по кадрове задання поворотів різноманітних кісток аніматором. Зазвичай ці методи поєднуються так що спочатку здійснюється запис анімацій за допомогою motion capture а згодом аніматори вручну допрацьовують анімації до потрібної якості і додають до них художні особливості.

Створення більшості анімацій потребує багато ручної роботи і цього ніяк не уникнути на даний момент, але існують ситуації, коли анімацію потрібно програти на скелеті із відмінним скелетом або додати до анімації невелику деталь, і доводиться переробляти цю анімацію ледь не з нуля. Саме тому було запропоновано розробити систему, що дозволила б уніфікувати анімації на концептуально схожих скелетах і додати процедурно генеровані деталі.

Було запропоновано використання методів вирішення задачі інверсної кінематики для розробки цієї системи. Інверсна кінематика дозволяє управляти кінцівками віртуальних скелетів не зважаючи на повну структуру кісток цих кінцівок, що робить їх можливим способом вирішення проблеми перенаправлення анімацій поміж персонажами із концептуально схожими віртуальними скелетами. Як приклад

концептуально схожих скелетів із реального світу можна розглянути скелети людини та орангутана. Зрозуміло, що ці два скелети мають однакові види кінцівок, такі як руки і ноги, але довжина цих кінцівок помітно відрізняється. Також ці скелети відрізняються можливими відносними позами кісток й іншими біомеханічними факторами, але інверсна кінематика могла би дозволити одній оригінальній анімації управляти персонажами із скелетами, що мають такі відмінності. Також тут постає важливою можливість створення процедурних ефектів для заданих анімацій, що може дозволити анімаціям після перенаправлення виглядати більш натуральними та реалістичними. Прикладом такого ефекту може бути маніпулювання руками персонажу.

Для реалізації системи на основі методів інверсної кінематики необхідно розробити проміжний формат анімацій, не прив’язаний до певного віртуального скелету, а також розробити метод конвертації традиційної анімацій у цей проміжний формат. Після цього, для перегляду програшу цього проміжного формату, потрібно реалізувати декілька алгоритмів вирішення задачі інверсної кінематики, що дозволять анімації у проміжному форматі впливати на дійсний віртуальний скелет персонажу. Для того, щоб система була повноцінною, необхідна також реалізація експорту проміжного формату назад у традиційний. Також важливим є забезпечення користувача зрозумілим і гнучким графічним інтерфейсом.

Усі з вище зазначених задач були успішно виконані та поєднані у одне програмне рішення, використовуючи мову програмування C++ та графічний ігровий рушій Urho3D.

1. ПОСТАНОВКА ЗАДАЧІ СТВОРЕННЯ СИСТЕМИ ГЕНЕРАЦІЇ ТА УНІФІКАЦІЇ АНІМАЦІЙ

Тривимірна скелетна анімація має широкий спектр застосувань: кінематограф і мультиплікація, ігрова індустрія і робототехніка. Скелетна анімація представляє собою техніку в комп'ютерній анімації, в якій персонаж представляється у двох частинах: поверхневе зображення, яке використовується для його відображення, яке називається мешем, і ієрархічним набором взаємопов'язаних кісток, яке називається скелетом і використовувався для трансформації мешу [1]. Приклад можна бачити на рисунку 1.1.

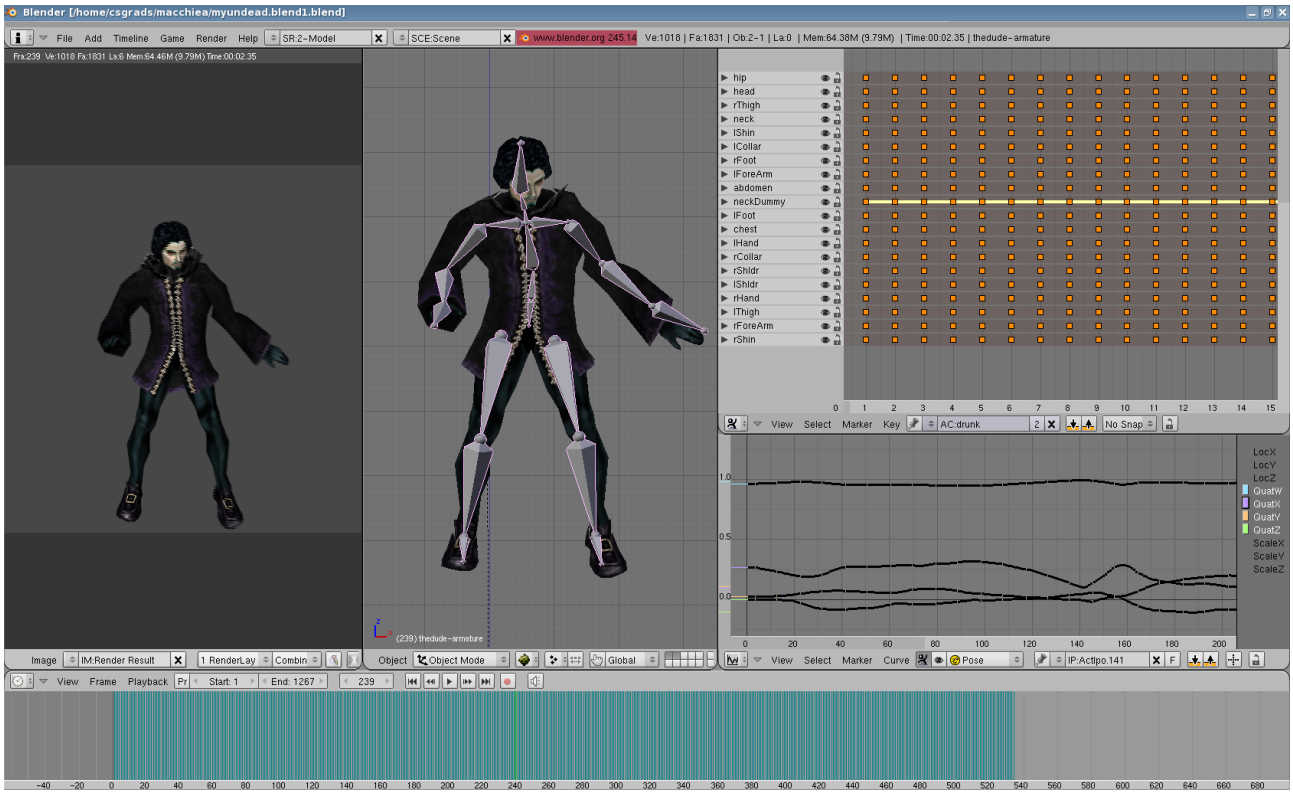

Рисунок 1.1 – Приклад скелету персонажа і його мешу

Хоча ця методика часто використовується для анімації людей або, в цілому, для органічного моделювання, вона слугує лише для того, щоб зробити процес анімації

більш інтуїтивним, і така ж техніка може бути використана для контролю деформації будь-якого об'єкта, наприклад, дверей, ложки, будівлі тощо.

Техніка скелетної анімації була введена в 1988 році Наді Магнітатом Талманом, Річардом Лаперірем і Даніелем Талманном[2]. Ця методика використовується практично у всіх системах анімації, де спрощені інтерфейси користувача дозволяють аніматорам контролювати рух персонажа за допомогою модифікування положення і орієнтації у просторі окремих кісток, а також з використанням технік запису руху або технік створення анімації за допомогою процедурних алгоритмів, таких як інверсна кінематика і фізична симуляція скелету і м’язів.

І хоча основна частина роботи аніматорів — створення анімацій для визначених персонажів, іноді постає задача перенесення анімацій поміж персонажами і це не є проблемою, якщо персонажі мають однаковий уніфікований скелет, але так буває не завжди. Якщо скелети персонажів сильно відрізняються, аніматорам доводиться вручну відтворювати великий об’єм роботи уже вкладений в анімацію.

На рисунку 1.2 можна бачити приклад персонажів із дуже різними скелетами, що робить неможливим використання на них спільних анімацій. З другого боку, їх скелети мають спільну рису — вони обидва прямоходячі, мають по дві ноги і дві руки, тобто схожу загальну структуру. Не дивлячись на це, використовуючи традиційні методи, потрібно було б повністю переробляти анімації аби їх застосувати. Саме на вирішення цієї проблеми направлене наше дослідження.

Як результат дослідження, потрібно створити програмний застосунок, основними функціями якого є:

— завантаження персонажів і їх традиційно створених анімацій.
— можливість програвати анімації на призначених для них персонажів.
— можливість конвертувати вибрану традиційну анімацію до проміжного формату.
— перегляд конвертованих анімацій на персонажах із концептуально-однаковим скелетом.
— додавання процедурних елементів до анімації.
— експорт анімацій у проміжному форматі і їх запис у традиційному форматі.

— забезпечення користувача зручним інтерфейсом для здійснення зазначених вище.

Рисунок 1.2 – Приклад персонажів з різними скелетами

2. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ УНІФІКАЦІЇ АНМАЦІЙ

З кожним роком з’являються все нові і нові інструменти створення тривимірної скелетної анімації засновані на різних принципах і ідеях, але проблема уніфікації анімацій створених для різних скелетів залишається в основному не вирішеною.

Саме тому було запропоновано до розробки саме систему уніфікації анімацій, яка дозволила б скоротити час, що витрачається аніматорами на рутинну роботу реплікацій анімацій поміж персонажами із різними скелетами і дозволити їм працювати над більш важливими речами, де потребується креативність.

2.1. Аналіз предметної області: створення скелету

Після створення 3D-моделі будується серія кісток, що представляють структуру скелета. Наприклад, у персонажі може бути група кісток спини, хребта і кістки голови. Ці кістки можуть бути трансформовані за допомогою програмного забезпечення для цифрової анімації, а саме: їхнє положення, обертання та масштаб. Записуючи ці аспекти кісток уздовж часової шкали (за допомогою процесу анімування по ключовим кадрам), анімації набувають своєї форми.

Простий базовий скелет може зайняти кілька годин або менше, а комплексний скелет для фільму може зайняти і кілька днів.[3]

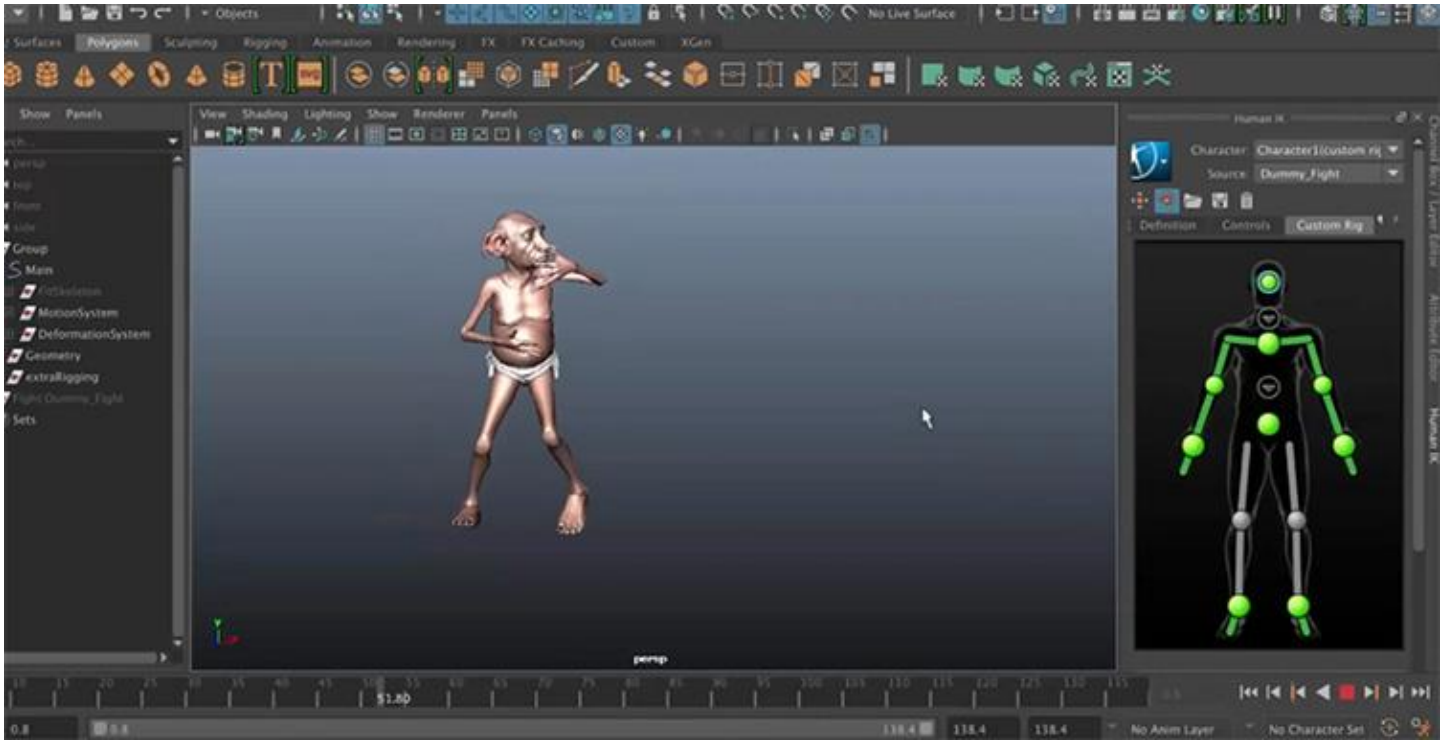

Рисунок 2.1 – Приклад персонажа і його ланцюжків кісток

Створення скелету призводить до ієрархічної структури, де кожна кістка знаходиться у відносинах батьків / дітей з кістками, з якими вона з'єднується. Це спрощує процес анімації в цілому. Коли аніматор рухає плечову кістку, передпліччя і кістки рук теж рухатимуться. Мета полягає в тому, щоб максимально точно імітувати реальне життя. Спосіб взаємодії 3D-моделі з кістками визначається за допомогою набору секторів ваги на меші. Вага контролює, на скільки впливає кістка на ділянку мешу. Таким чином, чутливість деформації може бути точно налаштована для точної анімації. Заповнення секторів із вагою кісток є невід'ємною частиною процесу створення скелету. Комп'ютер часто здатний автоматично генерувати ваги для моделі, але результат іноді не є пригідним до використання.

Розміщення кісток є найпростішою частиною створення скелету для моделі. Після розміщення багато кісток потребуватимуть додаткової роботи для можливості створення анімації належним чином.

Оснащення персонажа зазвичай вимагає від аніматора додавати інверсну кінематику до кісток. Інверсна кінематика використовується в основному для рук і ніг або інших кінцівок, таких як хвіст дракона. Хороша настройка інверсної

кінематики буде тримати лікті та коліна в правильному напрямку і дозволять аніматору легше досягти реалістичного руху.

2.2. Аналіз існуючих програмних засобів для уніфікації анімацій

У процесі пошуку інформації, та аналізу існуючих рішень для унфікації анімацій було виявлено, що існують деякі рішення для для схожої проблеми — перенаправлення анімацій, а саме в основному вони постачаються із двигунами для розробки ігор.

— Unity — кросплатформовий інструмент для розробки дво- та тривимірних додатків та ігор, що працюють на операційних системах Windows і OS X. У ньому присутня система перенаправлення анімацій у рамках скелету людини. Тобто структура скелету повинна бути ідентичною, можуть відрізнятися лише розміри і початкова орієнтація кісток;

— Unreal Engine — ігровий рушій, розроблюваний і підтримуваний компанією Epic Games. Його можливості по перенаправленню в основному співпадають з Unity, тобто можливе лише перенаправлення скелету людини.

Явним недоліком обох з цих рішень є неможливість використання інших видів скелету, окрім людського, тобто їх використання неможливе для скелетів четвероногих тварин або інших створінь.

Також недоліком можна вважати загальну схему роботи такої системи, адже вона лише калібрує повороти і переміщення кісток роблячи поправку на пропорції і не зможе допомогти, якщо скелети структурно відрізняються, як у прикладі на рисунку 1.2. Також, ці системи є частиною більших систем — ігрових рушіїв, що робить неможливим їх використання разом з іншими інструментами.

2.3. Висновки

Із зробленого аналізу можна зробити висновки, про складність реалізації універсальної системи перенаправлення анімацій. Це може бути можливою причиною відсутності на ринку відкритого і доступного рішення, а от відсутність попиту, можна виключити як можливий фактор, адже перенесення анімацій є задачею, з якою аніматори стикаються кожного дня. Також, цілком можливо, що деякі рішення цієї задачі існують, але лише у вигляді закритих технологій, розроблених у межах великих студій, що займаються розробкою ААА відеоігор. Розробка відкритого універсального програмного рішення задачі перенаправлення і переносу анімацій може мати велику цінність для декількох індустрій, включаючи індустрію кіно- та анімаційних фільмів та ігрову індустрію.

3. АНАЛІЗ АЛГОРИТМІВ ВИРІШЕННЯ ЗАДАЧІ ІНВЕРСНОЇ КІНЕМАТИКИ

Проблема інверсної кінематики довгі роки розглядалася науковцями в області технологій робототехніки та комп'ютерної графіка. Протягом останніх десятиліть була запропонована велика кількість різноманітних алгоритмів .

Так, як існує чимало алгоритмів вирішення задачі інверсної кінематики, надзвичайно важливо вибрати той, що найбільш підходить до формату задачі, для цього було обрано і проаналізовано основні алгоритми доступні на сьогоднішній день.

3.1 Детальний опис проблеми інверсної кінематики

Нехай повна спільна конфігурація скелету кінцівок робота чи персонажа визначається скалярами $\theta_{1,\dots}\theta_n$, припустимо, що існує n з'єднань, а кожне значення θ_j називається кутом спільного з'єднання (конфігурація суглоба може бути задана не лише кутом), де θj - кут в площині обертання, якщо ми знаємо усі осі обертання. Певні точки на зчленуваннях визначаються як кінцеві ефектори.

Щоб вирішити проблему інверсної кінематики, кути з'єднання повинні бути встановлені так, що отримана конфігурація кісток і суглобів розміщує кожен кінцевий ефектор у його цільовому положенні, або як можна ближче, до нього. Якщо існує k кінцевих ефекторів, нехай їх позиції позначаються як $s_{1,\dots}, s_k$ відносно деякого фіксованого положення. Кожна кінцева позиція ефектора S_i є функцією кутів суглобів. Вектор-стовпець $\left(s_{1,\dots}, s_k\right)^T$ може бути записаним як $\vec{S}$ і може розглядатися як вектор-стовпець із скалярними елементами $m = 3k$ або із k елементів із простору

R^3. Один із способів керування скелетом є визначення цільових позицій, по одній для кожного кінцевого ефектора. Цільові положення також можуть визначатися вектором $\vec{t} = (t_1, \dots, t_\kappa)^T$, де t_i є цільовим положенням для і-го кінцевого ефектора. Нехай $e_i = t_i - S_i$ - бажана зміна положення і-го кінцевого ефектора (переміщення до бажаної і-ї цілі). Це рівняння може бути переписано як $\vec{e} = \vec{t} - \vec{s}$.

Кути зчленування також можуть бути записані як вектор-стовпець $\theta = (\theta_1 \dots \theta_n)^T$. Кінцеві позиції ефекторів є функціями кутів суглобів; цей факт може бути виражений у рівнянні 3.1 як:

$$\vec{\mathbf{s}} = f(\boldsymbol{\theta}) \tag{3.1}$$

або, для $i = 1 \dots k,\ s_i = f_i(\theta)$. Це є рішенням задачі прямої кінематики.

Мета інверсої кінематики полягає в тому, щоб знайти вектор θ такий, що $\vec{s}$ дорівнює заданій бажаній конфігурації $\overrightarrow{s_d}$:

$$\boldsymbol{\theta} = f^{-1}(\vec{\mathbf{s}}_d) \tag{3.2}$$

де f - надзвичайно нелінійний оператор, який важко інвертувати.

Однак існують випадки, коли вирішення проблеми інверсної кінематики не існує через недосяжність цільових положень або оптимальне рішення не є унікальним. Навіть у ситуаціях із простими конфігураціями суглобів, виведення вищезазначених рівнянь може бути неможливим або майже неможливим. Тому використання ітеративних методів наближення оптимального рішення проблеми є необхідними. Надалі у цьому розділі ми будемо розглядати саме ітеративні методи.

3.2 Алгоритми на основі Якобіана

Якобіан J являється матрицею часткових похідних всієї системи кісток у ланцюжку відносно кінцевого еффектору s. Одна з перших пропозицій ефективного

методу розрахунку якобіану була запропонована Оріном і Шрадером у 1984 році[11]. Рішення на основі якобіану є лінійними наближеннями рішення задачі інверсної кінематики (як можна бачити на рисунку 3.1). J визначається як функція від θ - параметрів системи кісток.

$$J(\boldsymbol{\theta})_{ij} = \left(\frac{\partial \mathbf{s}_i}{\partial \theta_j}\right)_{ij} \tag{3.3}$$

А j-тий елемент якобіана визначається як

$$\frac{\partial \mathbf{s}_i}{\partial \theta_j} = \mathbf{v}_j \times (\mathbf{s}_i - \mathbf{p}_j) \tag{3.4}$$

де p – позиція j-тої кістки а v – одиничний вектор направлений вздовж осі обертання кістки. Варто зауважити, що J можна розглядати як матрицю $k \times n$, з елементами-векторами простору R^3 або як матрицю $m \times n$ з скалярними елементами $(m = 3k)$.

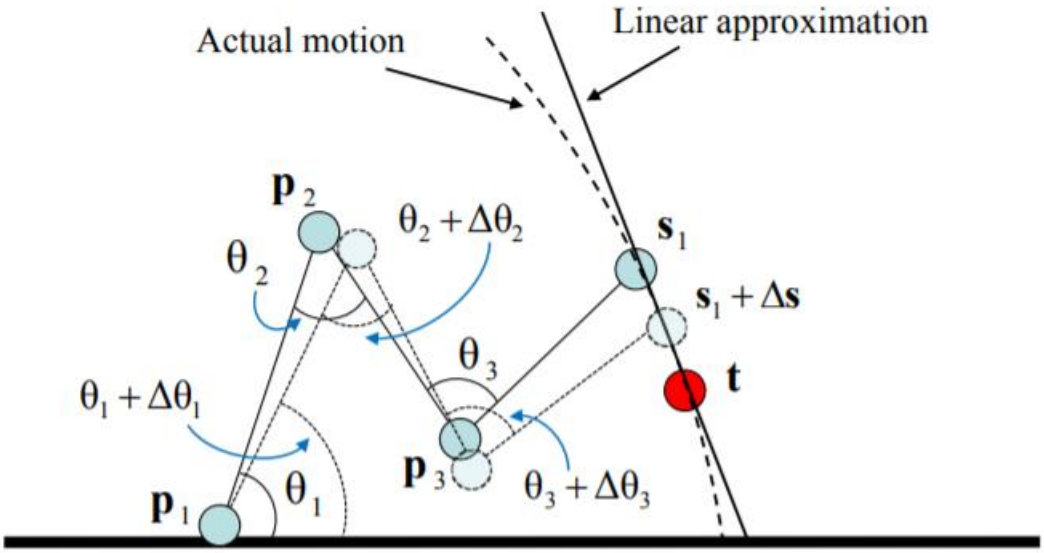

Рисунок 5.1 — ілюстрація лінійного наближення рішення за допомогою Якобіану

Використовуючи поточні значення θ, s_i і t_i, якобіан $J = J(\theta)$ можливо розрахувати для подальшого пошуку значення оновлення dθ з метою збільшення кутів з'єднання θ на $\Delta\theta$

$$\boldsymbol{\theta} := \boldsymbol{\theta} + \Delta\boldsymbol{\theta} \tag{3.5}$$

Зміни в кінцевих позиціях ефекторів, викликані цією зміною кутів з'єднань, можуть бути оцінені як

$$\Delta\vec{\mathbf{s}} \approx J\Delta\boldsymbol{\theta} \tag{3.6}$$

Ідея полягає в тому, що значення $\Delta\theta$ слід вибирати так, щоб $\Delta\vec{s}$ було приблизно рівним $\vec{e}$, хоча також загальноприйнято вибирати $\Delta\theta$ так, щоб приблизний рух ефекторів (частково) відповідав швидкостям цільових позицій у просторі.

З цього випливає, що рівняння прямої і інверсної кінематики можуть бути записані так:

$$\vec{\mathbf{e}} = J\Delta\boldsymbol{\theta} \tag{3.7}$$

$$\Delta\boldsymbol{\theta} = J^{-1}\vec{\mathbf{e}} \tag{3.8}$$

Рішення на основі Якобіану працюють не завжди добре по причині того, що зазвичай рішення не унікальне, що призводить до неможливості розв'язання Якобіану або його зведення в сингулярність, тобто рішення, яке насправді не рухає систему кісток.

На протязі років після оригінального дослідження, були запропоновані методи покращення такого рішення задачі інверсної кінематики. Такими покращеннями були:

— Псевдо-інверсний якобіан, також відомий, як інверсний якобіан Мура-Пенроуза. У цьому методі знаходиться інверсія якобіану, така, що

$$\Delta\boldsymbol{\theta} = J^{\dagger}\vec{\mathbf{e}} + (I - J^{\dagger}J)\varphi \tag{3.9}$$

Цей метод часто обговорюється, як один із простих рішень задачі інверсної кінематики, але рідко застосовується на практиці, по тим-же причинам, що і звичайний якобіан.

— Транспонований якобіан. Основна ідея методу транспонованого якобіану полягає у покращенні ефективності завдяки використанню транспонування якобіану, а не його інверсії, тобто

$$\Delta\boldsymbol{\theta} = \alpha J^T \vec{\mathbf{e}} \tag{3.10}$$

де a – скаляр, що підбирається для певного скелету. Цей метод був вперше використаний Балестріно у роботі у якій вирішувалась проблема управління роботичними маніпуляторами[12].

3.3 Ньютонівські методи

Сім'я методів Ньютона ґрунтується на розширенні ряду Тейлора другого порядку для такої функції f(x) що:

$$f(x+\sigma) \approx f(x) + [\nabla f(x)]^T \sigma + \frac{1}{2}\sigma^T H_f(x)\sigma \tag{3.11}$$

де $H_f(x)$ – матриця Гессе. Проте розрахунок матриці Гессе дуже складний і приводить до високих обчислювальних витрат для кожної ітерації. Було запропоновано декілька підходів, які замість розрахунку матриці Гессе використовують апроксимацію матриці Гессе на основі значення градієнта функцій. Найбільш відомими методами є метод Бройдена, метод Пауелла, а також метод Бройден, Флетчер, Голдфарб і Шанно (BFGS) [13, 14].

Оскільки методи Ньютона задаються як завдання мінімізації, вони повертають плавний рух без нерівномірних розривів. Також дуже просто застосовувати спільні обмеження поворотів кісток. Найбільш очевидним методом для запровадження обмежень є метод градієнтної проекції. Методи Ньютона також мають ту перевагу, що вони не страждають від проблем сингулярності, таких як ті, що виникають при пошуку зворотнього Якобіану. Однак їх недоліками є їх висока складність, важкість

реалізації і висока обчислювальна вартість кожної ітерації, що робить майже неможливим їх використання у застосунках реального часу.

3.4 Послідовний метод Монте-Карло

Відносно нещодавно для розв'язання задачі інверсної кінематики був запропонований послідовний метод Монте-Карло (SMCM). Корті і Арнауд[15] запропонували таке рішення на основі принципу вибірки даних із записаних анімцій. Використовуючи підхід вибірки, задача інверсної кінематики може бути вирішена на основі прямої кінематки, отже можна уникнути чисельної інверсії рівняння прямо кінематики, що допомагає методу зберігати стабільність. Задача складається як прихована Марківська модель (HMM), прихований стан якої задається всіма параметрами, що визначають фігуру, що буде управлятися алгоритмом. Отже, простір варіантів складається з усіх можливих конфігурацій стану. Інверсна кінематика потім переформулюється в рамках фільтрації результату роботи моделі. Запропонований вирішувач SMCM IK не вимагає явної числової інверсії, а спільні обмеження можуть бути додані в систему інтуїтивними методами. Вони можуть бути легко реалізовані без необхідності комплексних алгоритмів оптимізації. Недоліком методу є необхідність великої кількості даних для навчання алгоритму, що не завжди є можливим.

3.5 Циклічно-координатний спуск

Циклічно-координатний спуск — популярний евристичний ітеративний алгоритм, який підходить для інтерактивного управління тілом персонажа.[9] Він простий в реалізації і в сотні разів швидший ніж методи на основі Якобіану. Його ідея

заключається в максимальному наближенні кінцевого ефектору до цільової позиції при проході по кожній кістці в ланцюжку, як можна бачити в прикладі на рисунку 3.2.

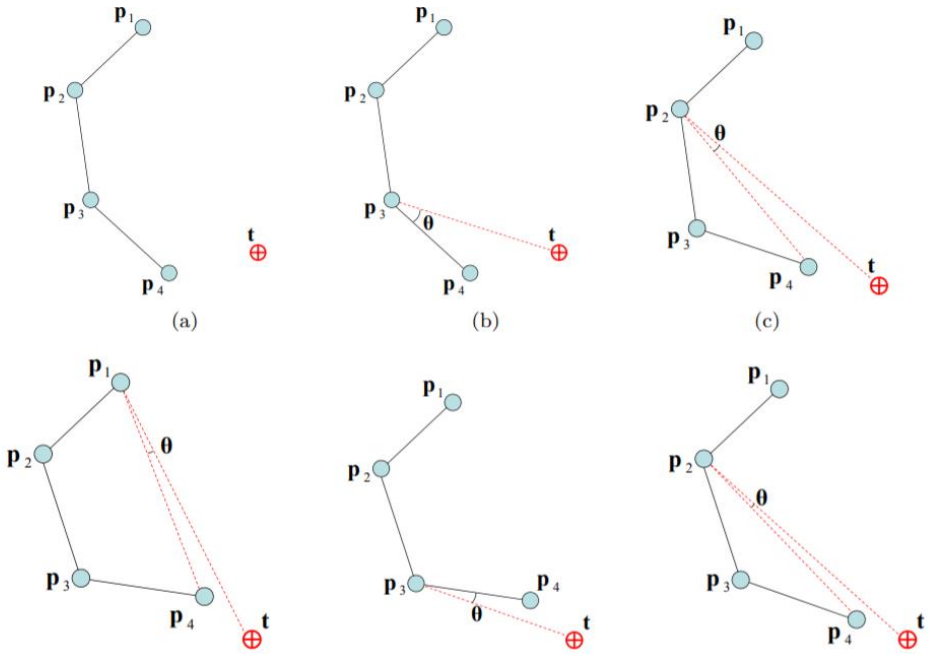

Рисунок 3.2 — ілюстрація роботи циклічно-координатного спуску

CCD є дуже простим у реалізації та надзвичайно швидким. Він забезпечує чисельно-стабільне рішення і має лінійну часову складність в залежності від кількості ступенів свободи. Метод CCD намагається мінімізувати помилки позиції та орієнтації шляхом перетворення перетворення однієї змінної за одну ітерацію. Алгоритм стверджує, що, починаючи від кінцевого ефектора всередину до основи маніпулятора, кожне з'єднання повинне бути перетворене так, щоб перемістити кінцевий ефектор як можна ближче до цілі. Цю процедуру повторюють до отримання задовільного результату. Обчислювальні витрати для кожного суглоба низькі, і тому рішення може бути сформульовано дуже швидко.

Як і інші зворотні кінематичні алгоритми, ЦЦС може згенерувати багато різних результуючих поз для заданої початкової пози. Задача вибору оптимального рішення з усіх згенерованих є нетривіальною, тому обмеження поворотів суглобів маніпулятора повинні бути включені для покращення оптимальності результуючої пози, яку повертає алгоритм. У ЦЦС легко застосувати локальні обмеження, варто просто обмежувати поворот кістку у бажаній площині і з бажаним скалярним

значенням після кожної ітерації, у такому випадку, якщо рішення існує, алгоритм гарантовано його знайде. Запровадження глобальних обмеження маніпулювання кістками є складною задачею для цього алгоритму, що не має рішення у загальному випадку.

Також до мінусів ЦЦС можна віднести ненатуральність кінцевих поз без застосування обмежень і складність реалізації ланцюжків на декілька ефекторів відразу.

3.6 FABRIK

FABRIK (Forwards And Backwards Reaching Inverse Kinematics) – новітній швидкий алгоритм рішення задачі інверсної кінематики представлений у 2010 році[10]. Метод полягає у пересуванні кісток із збереженням відносних довжин до кінцевої точки і назад до точки початку, він не використовує кути і повороти у своїй роботі, а лише вектори. Роботу алгоритму можна описати так: нехай у нас є позиції кісток включаючи позицію базової кістки і позиція кінцевого ефектору, тоді, пересунемо кінцеву кістку на місце ефектора, наступну кістку пересунемо так, щоб не порушувалась відстань між кістками, і так до останньої базової кістки, після цього повторюємо все знову, лише починаючи з базової кістки пересуваємо її в початок і ідемо в зворотньому напрямку, як можна бачити на рисунку 3.3.

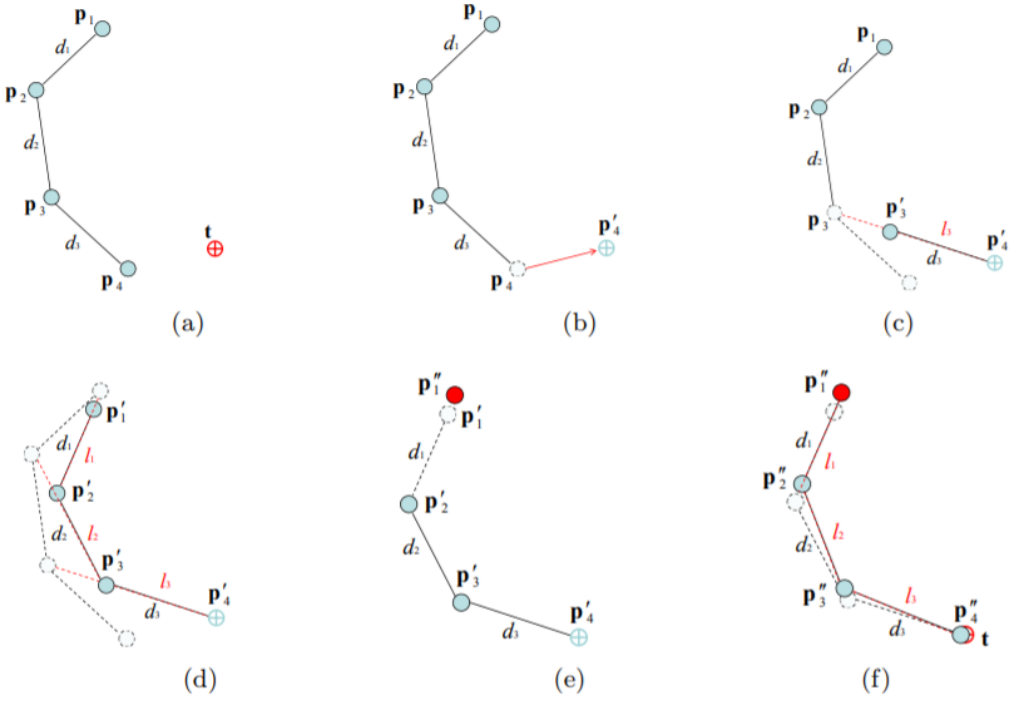

Рисунок 3.3 — приклад роботи FABRIK

3.6.1 Опис алгоритму

Припустимо, що $P_1 \dots P_n$ – позиції суглобів(кісток) маніпулятора. Крім того, припустимо, що P_1 – базова кістка(з'єднання), а P_n – кінцевий ефектор, для простого випадку, коли існує тільки один кінцевий ефектор. Цільову позицію позначимо як t, а початкове базове положення - b. Графічне представлення повної ітерації алгоритму з єдиною мішенню і 4 суглобами можна бачити на рисунку на малюнку 5.3. Спочатку обчислимо відстані між кожним суглобом $d_i = |p_{i-1} - p_i|$, для $i = 1, \dots, n-1$. Після цього, перевіримо, чи цільова позиція досяжна чи ні: знайдемо відстань між коренем і ціллю, і якщо ця відстань менше, ніж загальна сума всіх довжин кісток – рішення існує.

Повна ітерація складається з двох етапів. На першому етапі алгоритм оцінює кожне положення суглоба, починаючи з кінцевого ефектора, p_n, переміщаючись

всередину до бази маніпулятора, p_1. Отже, нехай нове положення кінцевого ефектора є цільовим положенням, $p'_n = t$. Знайдемо лінію, l_{n-1}, яка проходить через спільні позиції p_{n-1} та p_n. Нове положення *(n - 1)* -го суглоба, p_{n-1}, лежить на цій лінії з відстанню d_{n-1} від p_n. Аналогічно, нове положення *(n − 2)* -го суглоба, $p'_{(n-2)}$, можна обчислити за допомогою лінії l_{n-2}, яка проходить через p_{n-2} і p'_{n-1}, а відстань d_{n-2} від p'_{n-1}. Алгоритм продовжується для всіх нових позицій кісток, включаючи кореневу, p'_1. Маючи на увазі, що нове положення кореня маніпулятора c'_1, не повинне відрізнятися від початкового положення, необхідний другий етап алгоритму. Повна ітерація завершена коли повторюється та ж процедура, але цього разу, починаючи з кореневого суглоба і рухаючись назовні до кінцевого ефектора. Таким чином, нехай нове положення для 1-го суглоба, c''_1, буду його початковим положення b. Потім, використовуючи лінію l_2, яка проходить через точки p''_1 і p'_2, ми визначаємо нове положення суглоба p''_2 як точку на цій лінії з відстанню d_1 від p''_1. Ця процедура повторюється для всіх інших кісток, включаючи кінцевий ефектор. У випадках, коли корінний суглоб повинен бути переведений в потрібне положення, FABRIK працює так, як описано з тією різницею, що у зворотній фазі алгоритму нове положення кореневого суглоба, p"1, буде у заданій бажаній позиції, а не початкової позиції.

Після закінчення однієї повної ітерації, кінцевий ефектор зажвди ближче до цільової позиції. Потім процедуру повторюють, настільки багато разів, наскільки потрібно багато, поки позиція кінцевого ефектору ідентична або досить близька до бажаної мішені. FABRIK завжди збігається до будь-яких заданих ланцюгів / цільових позицій, коли ціль знаходиться в межах досяжності. Якщо існують обмеженнями, які не дозволяють ланцюгу згинатися під певними кутами і це робить неможливим досягнення цільової позиції, або повної довжини ланцюжка недостатньо, щоб досягти цілі, існує умова припинення: порівнюються попередня і поточна позиція кінцевого ефектора, і якщо ця відстань менше, ніж певне мале число, FABRIK припиняє свою роботу. Також, в крайньому випадку, де кількість ітерацій перевищила вказане значення, а ціль не була досягнута, алгоритм припиняється.

Одна із простих можливих оптимізацій даного методу - побудова лінії, що вказує на цільову позицію, коли вона є недосяжної через недостатню спільну довжину кісток.

Цей метод має всі переваги існуючих ітераційних евристичних алгоритмів. Обчислювальна вартість для кожного спільного з'єднання на кожну ітерацію є низькою, що означає, що рішення знаходиться дуже швидко. Він також дуже легко реалізується, оскільки рішення включає у себе лише точки, відстані і лінії, і завжди повертає рішення, коли цільова позиція знаходиться в діапазоні досяжності. Він не вимагає складних обчислень (наприклад, якобіану або матриці Гессе) або матричних маніпуляцій, а також не страждає від проблем сингулярності і повертає плавний рух без нерівномірних розривів.

3.6.2 Застосування для декількох ефекторів

Вирішувачі проблем інверсної кінематики зазвичай використовуються для вирішення проблеми інверсної кінематики у багатьох областях. Більшість моделей з кількома ланцюжками кісток, такі як руки, тіло людини або ноги тощо, складаються з декількох кінематичних ланцюгів, і кожен ланцюг, як правило, має більше одного кінцевого ефектора. Тому для вирішувача важливо мати змогу вирішувати проблеми з кількома кінцевими ефекторами і цілями. Алгоритм FABRIK може бути легко розширений для обробки моделей з декількома кінцевими ефекторами. Алгоритм поділяється на два етапи, як і у випадку одного кінцевого ефктора. На першому етапі застосовується нормальний алгоритм, але цього разу, починаючи з кожного кінцевого ефектора і рухаючись всередину до з’єднання кількох ланцюжків. Це дозволить створити стільки різних позицій спільної базової кістки, скільки існує кінцевих ефекторів, пов'язаних з певною підгрупою кісток. Нова позиція цієї кістки буде центроїдом усіх цих позицій. Будемо називати таку кістку, що є базовою для декількох ланцюжків під-базовою. Після цього нормальний алгоритм повинен застосовуватися всередину, починаючи з цієї під-базової кістки, до самої кореневої

частини маніпулятора. Якщо існує більше під-базових кісток між попередньою під-базовою кісткою і коренем системи, та ж сама методика повинна використовуватися. На другому етапі застосовується нормальний алгоритм, починаючи від кореня і рухаючись назовні до під-базових кісток. Потім алгоритм повинен застосовуватися окремо для кожного ланцюжка до кінцевого ефектора; якщо існує більше під-базисів, застосовується той самий процес. Метод повторюється до тих пір, поки всі кінцеві ефектори не досягнуть мети або не відбудеться суттєвої зміни між їхніми попередніми та новими позиціями. Приклад моделі з кількома кінцевими ефекторами та декількома під-базовими кістками представлений на рисунку 3.4. Можна також використовувати більш складні моделі, розширюючи запропонований алгоритм, з урахуванням форми, обмежень і властивостей фігури, що дає більш швидкі і реалістичні результати. Такі моделі зменшують кількість ітерацій, необхідних для досягнення цілей і повернення більш реальних пози.

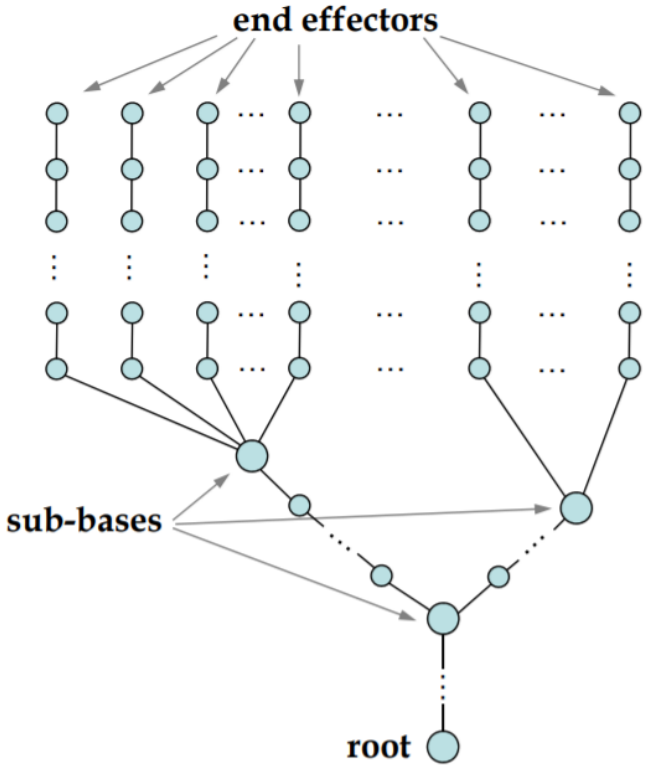

Рисунок 3.4 — приклад системи з кількома під-базовими кістками

3.6.3 Застосування із обмеженнями поворотів

Більшість моделей із ногами складаються із суглобів, що мають біомеханічні обмеження, які забезпечують природні обмеження їх руху. Такі обмеження є істотними в фізичному моделюванні і методах вирішення проблеми інверсної кінематики, оскільки вони дозволяють зменшити кількість візуально нереалістичних рухів.

У цьому розділі описана методологія включення обмежень маніпулятора при використанні FABRIK. Оскільки FABRIK є ітеративним, спільні обмеження можуть бути застосовані на кожному етапі просто шляхом обчислення поточної орієнтації і повернення її у межі допустимого діапазону. Здатність FABRIK сходитись до правильного результату, якщо мета знаходиться в межах досяжності, не залежить від будь-яких накладених спільних обмежень.

Основною ідеєю цієї методології є повторне позиціонування та переорієнтація цілі в межах допустимих діапазонів. Гарантія того, що ці обмеження завжди задовольняються, дозволяє досягти більш доцільної пози. Це може бути досягнуто шляхом перевірки, чи є ціль в межах допустимих меж, на кожному кроці FABRIK, і якщо це не так, щоб гарантувати, що вона буде переміщена відповідно. Припустимо, що ми маємо шарнірно-гніздовий шар з обертальними та орієнтаційними межами, обмежуючи дозволений простір до реалістичної підмножини. Її орієнтація описується кватерніоном r та її обертанням на кути $\theta 1, ..., \theta 4$. На рисунку 3.5 наведено графічне представлення реалізованих обмежень і нерегулярний конус, що описує межі обертального руху.

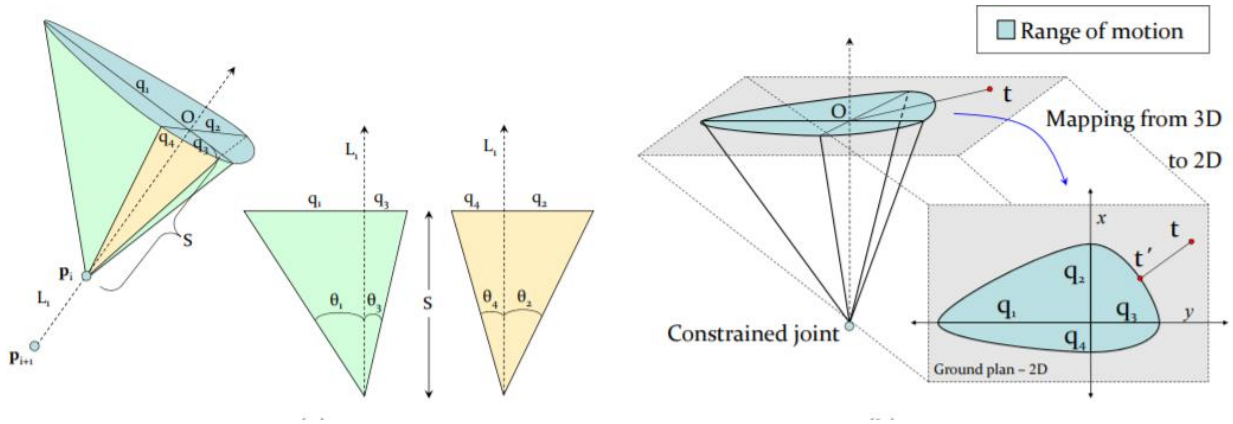

Рисунок 3.5 — приклад обмеження кістки у вигляді конусу

Орієнтацію суглоба можна знайти наступним чином: припустимо, що ми знаходимося на першому етапі алгоритму, тобто ми тільки що розрахували нове положення суглоба p'_i, і хочемо знайти нове положення p'_{i-1}. Необхідно знайти кватерніон, що виражає обертання між орієнтаційними рамками на суглобах p'_i та p_{i-1}, і якщо цей кватерніон являє собою обертання, що перевищує межу, переорієнтуємо кістку p_{i-1} таким чином, щоб обертання було в дозволених межах. Повторюємо процедуру для всіх суглобів на обох етапах алгоритму. Демонстрацію цього процесу можна бачити на рисунку 3.6.

Цей метод для обмеження суглобів застосовується на усіх етапах алгоритму і для кожної ітерації до досягнення мети або відсутності значних змін у положеннях кінцевих ефекторів на кожній наступній ітерації. Алгоритм може вирішувати обмеження суглобів та кінцівок, що мають 3 ступені свободи, і він може обробляти випадки повороту суглобів і кінцівок. Важливо нагадати, що довжини кісток не змінюються з плином часу, оскільки ці відстані залишаються незмінними при ітераціях FABRIK.

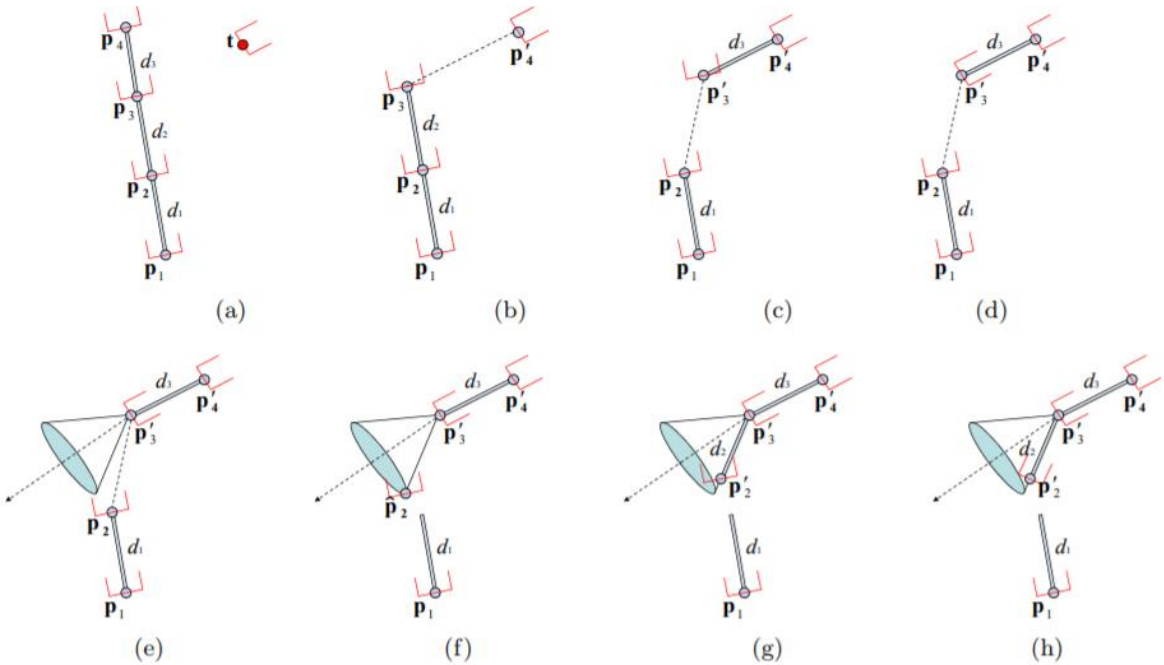

Рисунок 3.6 — приклад вирішення FABRIK із обмеженням

Мінусом FABRIK можна вважати погану підтримку обмежень других видів, окрім конічних, особливо його нестабільність при роботі з обмеженнями типу завіса.

3.7 Висновки

За результатом виконаного аналізу доступних алгоритмів вирішення задачі інверсної кінематики, було вирішено реалізувати ЦЦС і FABRIK. Ці алгоритми є сучасними і досить популярними методами вирішення такої задачі. Важливою характеристикою цих методів є їх стабільність і гарантій повернення результату за умови його існування, або повернення близького до можливого результату за умови неможливості вирішення задачі.

Було вирішено використовувати кожен з алгоритмів у найкращому для них сценарії, адже, як було показано вище, обидва алгоритми мають яскраво виражені плюси та мінуси, саме на них базується їх використання у програмній системі. Алгоритм FABRIK використовується для це ланцюжків хребта. Він забезпечує

стабільне і натурально-виглядаюче вирішення довгих ланцюжків кісток із застосуванням конічних обмежень. Циклічно-координатний спуск же використовується для ланцюжків рук і ніг, тому що саме ці частини тіла за своєю біомеханікою мають обмеження типу «завіса», а саме на кістках ліктів і колін, а як було показано у аналізі, цей метод дозволяє реалізувати такий вид обмежень досить просто, в той час, як у FABRIK, реалізація таких обмежень є неможливою і робить алгоритм нестабільним.

4. ЗАСОБИ РЕАЛІЗАЦІЇ ПРОГРАМНОЇ СИСТЕМИ

Аналізуючи поставлену задачу та методи її вирішення, було вирішено розроблювати програмний комплекс на основі одного із відкритих рушіїв відмальовування комп'ютерної графіки у форматі класичного застосунку для операційної системи Windows. Це дозволило зконцентруватись на вирішенні основних функціональних завдань і спростити їх імплементацію. Вибір операційної системи Windows як основної платформи зумовлено тим, що ця операційна система є стандартом де-факто у сучасних кіно- та ігрових- індустріях.

4.1. Опис архітектури застосунку

Для реалізації застосунку було використано ігровий рушій Urho3D, в основу якого покладено архітектуру ігрових рушіїв на основі вузлів(нод) із відношеннями дітей і батьків і компонент, які забезпечують поведінку нод. У загальному ця архітектура відповідає класичній entity-component архітектурі, у якій поняттю сутності у Urho відповідає поняття вузла. Така модель полегшує розробку тому, що дозволяє суттєво зменшити зв'язність між об'єктами у світі. На рисунку 4.1 можна бачити відносини між нодами і компонентами у такій архітектурі.

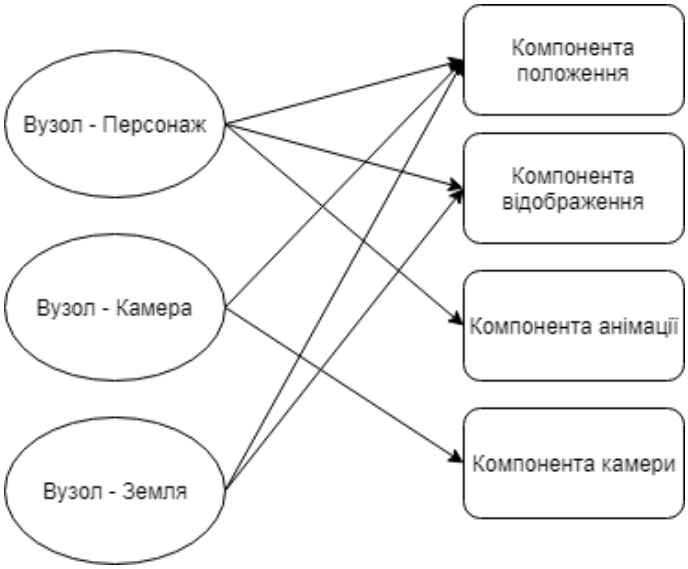

Рисунок 4.1 – Схема відносин вузлів і компонент

Окрім нод основу архітектури рушія складають контекст застосунку і система розповсюдження подій поміж нодами. Вона дозволяє просто реалізовувати реагування на будь-які події всіма зацікавленими модулями без явного зв’язування відправника події і тих, хто буде на неї реагувати, замість цього, контекст застосунку інкапсулює у собі систему відправки і підписки на певні повідомлення, а аргументи передаються через глобальну гетерогенну хеш-таблицю, ключами у якій є назви аргументів визначені окремо для кожної події. Схему роботи системи подій проілюстровано на рисунку 4.2.

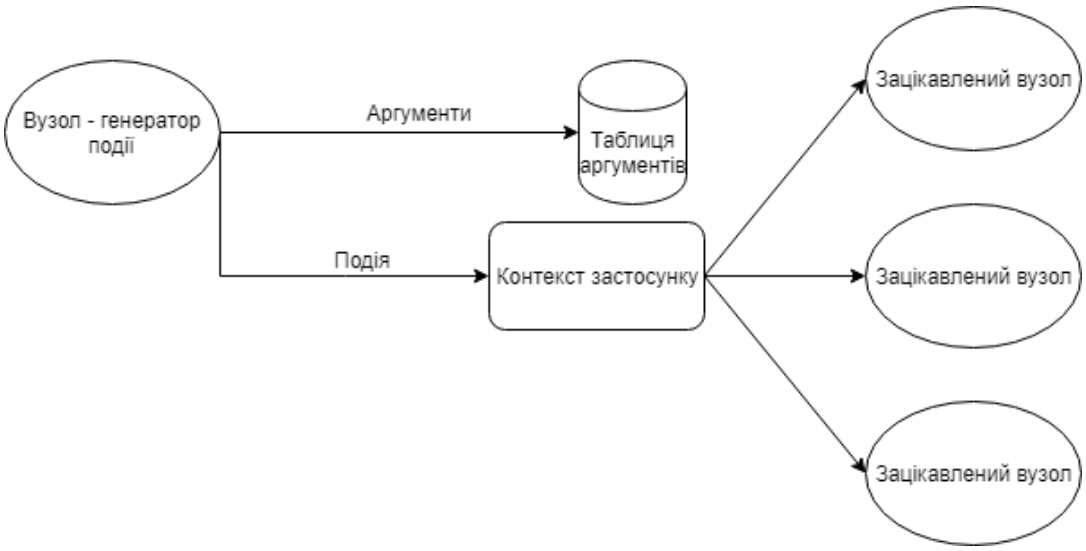

Рисунок 4.2 – Схема роботи системи подій

Така система розповсюдження подій є більш складною, але в той же час більш корисною і гнучкою у використанні реалізацією патерну Спостерігач. Спостерігач це поведінковий шаблон проектування що реалізує у класу механізм, який дозволяє об'єкту цього класу отримувати оповіщення про зміну стану інших об'єктів і тим самим спостерігати за ними [4].

Великі частини рушія реалізовані за допомогою так званих підсистем, доступ до яких є у кожного вузла системи. Прикладом підсистеми може бути підсистема користувацького інтерфейсу або підсистема рендеру.

4.2 Опис інструментів розробки

Програмна система повністю побудований, як класичний застосунок для операційної системи Windows основі графічного рушія Urho3D. Urho3D є безкоштовним, легким у плані ресурсів машини і використанні, крос-платформенним 2D і 3D ігровим рушієм, реалізованим на мові C++ і випущеним під ліцензією MIT. Він черпає багато ідей і натхнення від таких відкритих ігрових рушіїв як OGRE і Horde3D.

Увесь застосунок реалізовний на мові програмування C++, так як вона є основною мовою рушія Urho3D. Варто зазначити що він надає також API для таких мов програмування як AngelScript і Lua, але вони не використовувалися для розробки, по причині прагнення досягнути максимальної ефективності рішення.

C++ - це компільована, статично типізований мова програмування загального призначення. Підтримує такі парадигми програмування, як процедурне програмування, об'єктно-орієнтоване програмування, узагальнене програмування. Мова має багату стандартну бібліотеку, яка включає в себе поширені контейнери і алгоритми, введення-виведення, регулярні вирази, підтримку багатопоточності і інші можливості. C++ поєднує властивості як високорівневих, так і низькорівневих мов. [5][6] У порівнянні з його попередником - мовою C, - найбільшу увагу приділено підтримці об'єктно-орієнтованого і узагальненого програмування. [6]

C++ широко використовується для розробки програмного забезпечення, будучи одним з найпопулярніших мов програмування. Область його застосування включає створення операційних систем, різноманітних прикладних програм, драйверів пристроїв, додатків для вбудованих систем, високопродуктивних серверів, а також ігор. Існує безліч реалізацій мови C++, як безкоштовних, так і комерційних і для різних платформ. Наприклад, на платформі x86 це GCC, Visual C++, Intel C++ Compiler, Embarcadero (Borland) C ++ Builder і інші.

Синтаксис C++ успадкований від мови C. Одним з принципів розробки було збереження сумісності з C. Проте, C++ не є в строгому сенсі надмножиною C; множина програм, які можуть однаково успішно транслюватися як компіляторами C, так і компіляторами C++, досить велика, але не включає всі можливі програми на C.

Для розробки використовувалась остання доступна стабільна версія мови, а саме — C++ 17, що дозволило забезпечити кращу безпеку і стабільність програми, а також більш ефективно реалізувати ядро системи завдяки перевагам, які надає сучасна версія мови.

В якості компілятора і середовища розробки використовувалась інтегроване середовище розробки Visual Studio 2017 Community Edition. Visual Studio була

вибрана в якості основного середовища, оскільки вона є стандартом де-факто у розробці застосунків для Windows.

4.3 Обґрунтування вибору програмної реалізації

При проектуванні системи було вивчено та проаналізовано предметну область та вимоги які ставляться в умовах сучасного процесу створення анімаційних продуктів. Після ретельного аналізу було вирішено розроблювати програмний продукт, заснований на готовому відкритому рушію рендеру.

Вибір рушія здійснювався поміж декількох варіантів, серед них були такі опції:

— bgfx – бібліотека, яка забезпечує API, що дозволяє абстрагуватися від конкретного графічного API платформи, такого як DirectX або OpenGL. Інтерфейс, який вона постачає, все ще дуже близький до низькорівневого API платформи, тому було вирішено, що реалізація системи на основі цієї бібліотеки займе занадто багато часу, ніяк не пов’язаного із реалізацією головного функціоналу системи.

— Ogre3D – об’єктно-орієнтований рушій комп’ютерної графіки. Він орієнтований на роботу із сценою, гнучкий, написаний на C++. Розроблений, щоб бути легким і інтуїтивно зрозумілим для розробників для створення програм, що використовують апаратно-прискорену 3D-графіку. Не був вибраний через свій занадто “роздутий” інтерфейс, що робив роботу з окремими кістками надзвичайно важкою, тож його відкинули, як варіант.

— Urho3D - безкоштовний, легкий у плані ресурсів машини і використання, крос-платформенний 3D-ігровий рушій, реалізованим на C++ і випущений під ліцензією MIT.

У результаті аналізу був вибраний саме Urho3D, як найбільш вдалий інструмент для поставленої задачі.

4.4 Висновки

У результаті аналізу предметної області і підбору методів програмної реалізації, рішення було імплементовано із класичною архітектурою game object – component. Головною відмінностями стали архітектурні особливості використаного рушія Urho3D, а саме:

— Використана система «нод», а не «об’єктів»;
— Поведінка кожної «ноди» на сцені забезпечена реалізацією відповідної компоненти, що дозволило підвищити гнучкість такого рішення;
— Додаткова гнучкість архітектури, а також легкість наступної підтримки системи забезпечується реалізацією системи розповсюдження подій, що підтримує як прямий виклик обробників певної події, так і асинхронне оброблення подій.

Також максимальна ефективність програмної реалізації забезпечилась використанням сучасного діалекту мови програмування C++, а саме C++17. Використання низькорівневої мови програмування для реалізації всього функціоналу відобразилось у швидку і зручну у використанні програму, чого, можливо, неможливо було б досягти, використовуючи вбудовані можливості використання мови написання сценаріїв Lua.

5. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Система уніфікації анімацій складається чотирьох головних частин. Перший із них — конфігуратор проекту, у якому користувач задає персонажів і їх початкові анімації, з якими буде вестись робота. Наступна частина — система перегляду традиційних анімацій. Третя частина — генератор проміжного формату, який бере традиційну анімацію і перетворює її у проміжний формат на основі інверсної кінематики. Четверта і остання підсистема — програвач проміжного формату і експортер у традиційний формат.

На рисунку 5.1 наведена схема структури системи, на якій розташовані всі програмні модулі.

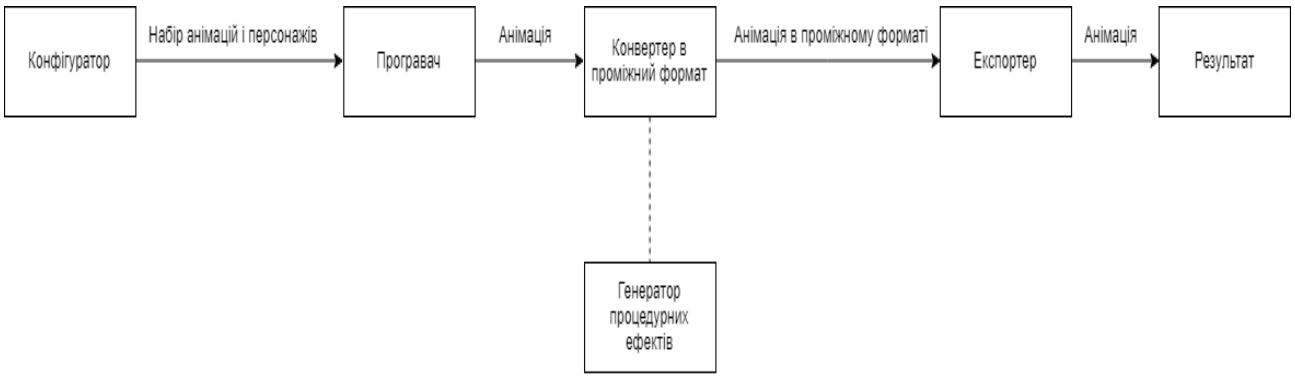

Рисунок 5.1 — Схема структури системи

5.1 Опис функціональності системи

Програмний застосунок для уніфікації анімації містить у собі одного головного актора – користувач системи, аніматора;

На рисунку 5.2 представлена діаграма прецедентів, яка описує функції та дії актора у системі.

Рисунок 5.2 — Діаграма прецедентів системи

5.2 Метод завантаження і програшу анімацій

Анімації зберігаються у форматі *.ani. Цей формат представляє собою бінарний файл без компрессії, у якому послідовно зберігаються трансформи усіх кісток.

Трансформ представляє собою набір із вектора, який позначає позицію, кватерніона, який позначає поворот, і вектора який позначає розміри кістки у просторі.

Після завантаження у пам’ять вони представляють собою об’єкт класу Animation, що складається з треків, які є анімаціями кожної окремої кістки у її локальному просторі. У свою чергу кожен трек складається з кадрів, які являють собою повну інформацію про положення кістки у даний момент часу. Діаграму цього відношення можна бачити на рисунку 5.3.

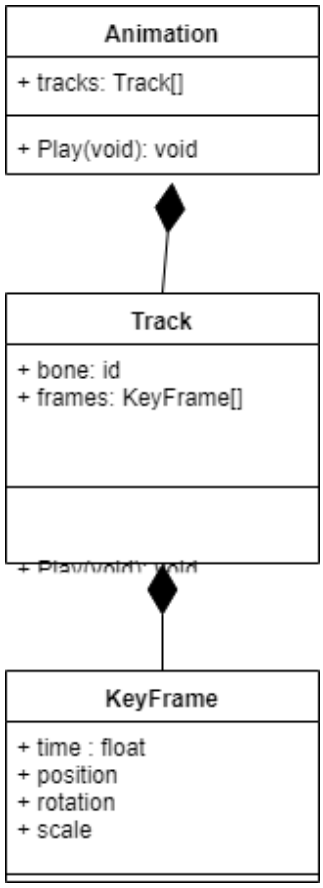

Рисунок 5.3 — Діаграма класу Animation

При програші кісткам встановлюються відповідні значення кадрів анімації, відповідно до поточного часу програшу анімації. Для того, щоб знайти фінальну позицію у світових координатах кожної вершини мешу, яким управляють кістки, вирішується задача прямої кінематики.

Пряма кінематика це процес визначення положення кінцевого положення залежних кісток при різних заданих параметрах батьківських кісток. Пряма кінематика активно використовується в робототехніці. Її математичне визначення можна записати так: кінематичні рівняння для ланцюга з’єднань кінцівки маніпулятора робота отримані з використанням жорсткої трансформації [Z] для характеристики відносного переміщення, дозволеного на кожному з'єднанні і окремої жорсткої трансформації [X] для визначення розмірів кожної ланки. В результаті виходить послідовність жорстких перетворень з'єднань і перетворення положень ланцюга від основи ланцюга до його кінцевої ланки, яка прирівнюється до заданої позиції для кінцевого ланки, як це можна бачити у формулі 5.1:

$$[T] = [Z_1][X_1][Z_2][X_2]\dots[X_{n-1}][Z_n], \tag{5.1}$$

де [T] є положенням у просторі останньої ланки. Ці рівняння називаються кінематичними рівняннями послідовного ланцюга.[7] Також рівняння прямої кінематики можна спрощено записати так, як показано у рівнянні 5.2

$$[T] = {}^{0}T_n = \prod_{i=1}^{n} {}^{i-1}T_i(\theta_i) \tag{5.2}$$

де θ_i — параметр(поворот і позиція) і-тої кістки.[8]

5.3 Проміжний формат анімацій

Для уніфікації анімацій нам потрібно спочатку представити їх у вигляді проміжного формату, який потім можна адаптувати для кожного персонажу окремо автоматично. Для цього введемо поняття інверсного скелету. Це скелет, який задається набагато меншою кількістю кісток ніж звичайний. Він визначає ланцюжки кісток, з якими буде працювати алгоритм уніфікації і визначається для кожного типу

персонажу, такого як, наприклад, прямо ходячі двоногі персонажі. На рисунку 5.4 можна бачити порівняння повноцінного скелету людини і інверсного скелету.

Можна бачити, що інверсний скелет визначає у цьому випадку ланцюжки з двох кісток для рук, ніг і хребта. Ці кістки потім будуть співвідноситись з кістками в скелетах реальних персонажів.

Проміжний формат представляє собою просто позиції кінців ланцюжків, наприклад долонь, у координатах початків ланцюжків, у цьому випадку плечей, а також загальну довжину ланцюжка для пристосування анімації до персонажів з різною довжиною кінцівок.

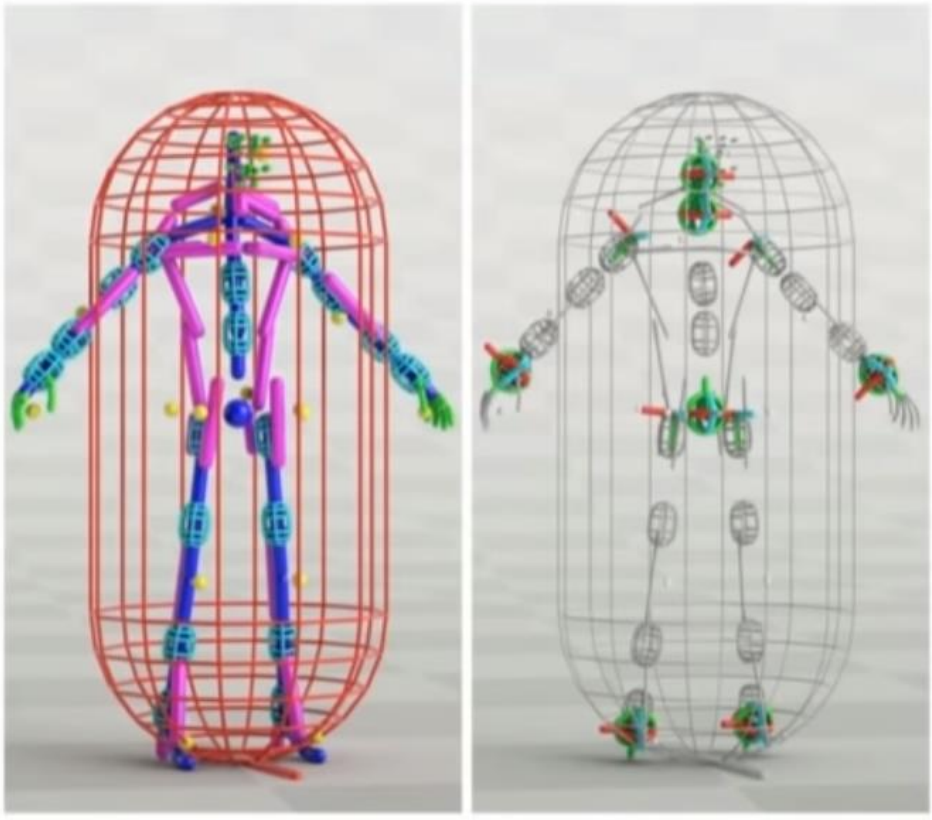

Рисунок 5.4 — порівняння повного скелету(зліва) і інверсного скелету

5.4 Програвання проміжного формату і експорт

Для того, щоб відтворити анімацію, записану у вигляді проміжного формату, нам потрібно знайти параметри кісток при відомому положенні кінцевих кісток, тобто ми вирішуємо задачу протилежну прямій кінематиці. Вона називається інверсною кінематикою.

Експорт анімацій в традиційний формат відбувається так: програється анімація у проміжному форматі, записуються всі параметри кісток на кожному кадрі у структуру анімації, описану вище, після чого вона може бути збережена у бінарному форматі. Після цього з нею можна працювати і в інших інструментах.

6. МЕТОДИКА РОБОТИ КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

Розроблена програмна система розроблена для використання з операційною системою Windows 10, так як саме ця система зазвичай використовується при роботі аніматорів.

6.1 Інсталяція та системні вимоги

Для запуску програми потрібна операційна система Windows 10. Потрібно запустити інсталятор програми, перед цим переконавшись, що у користувача установлені актуальні версії драйверів відеокарти, так як програма працює з апаратно-прискореним 3Д. Мінімальні вимоги до комп’ютера:

— Intel Core i3 6006U
— Nvidia GeForce MX150
— GB RAM
— 500 MB Storage

6.2 Інструкція з використання програмного продукту

При вході в систему користувач повинен авторизуватися і обрати проект, або створити новий проект і нового користувача. Це показано на рисунку 6.1.

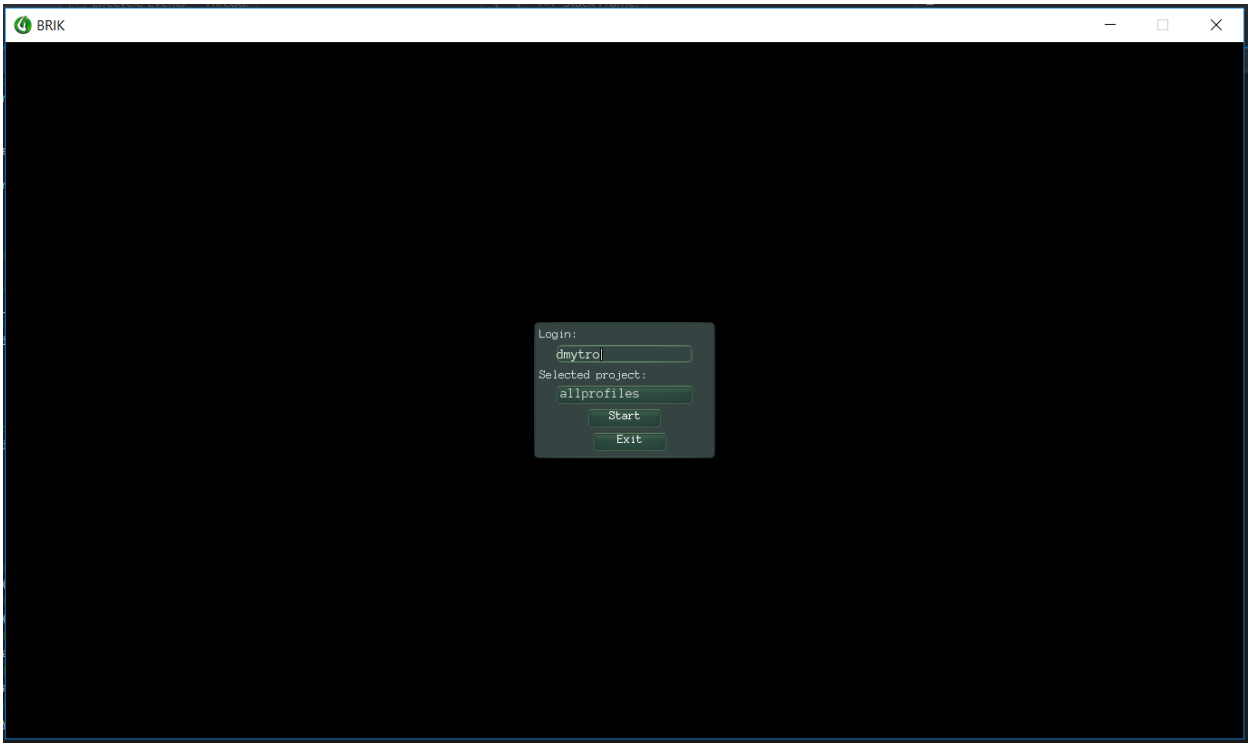

Рисунок 6.1 — Форма авторизації

Після того, як користувач авторизувався в системі, він отримує доступ до конфігуратора проекту(рисунок 6.2) і зможе змінити персонажів, додати анімації тощо.

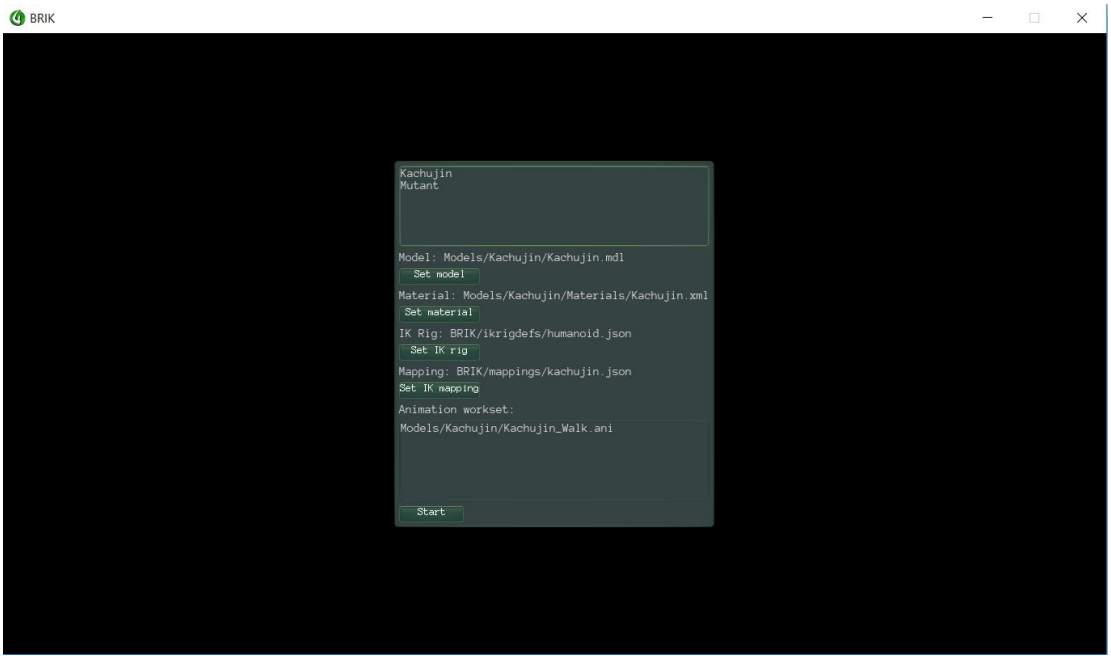

Рисунок 6.2 — Конфігуратор проекту

Після цього користувач може потрапити до основного вікна програми(рисунок 6.3), де він має змогу переглядати готові анімації, оглядати персонажів, конвертувати анімації в проміжний формат, переглядати їх і експортувати їх назад до традиційного формату.

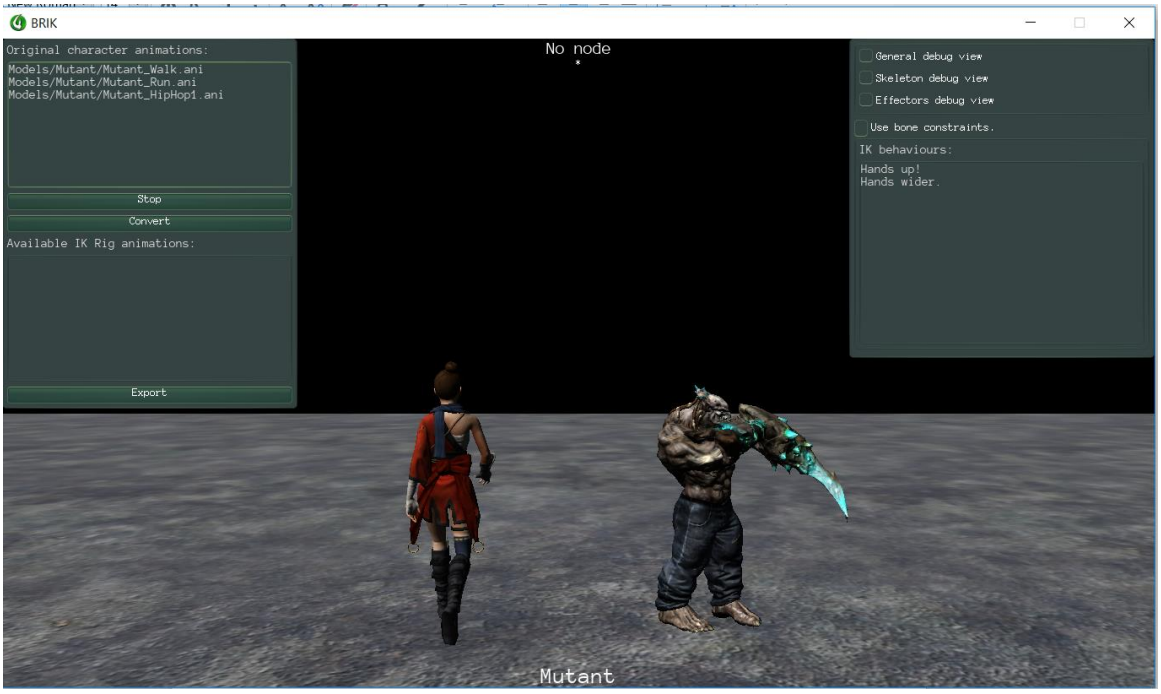

Рисунок 6.3 — Головне вікно програми

Також справа в користувацькому інтерфейсі представлені перемикачі різних режимів виду для можливості більш зручної роботи із кістками персонажів і ефекторами кісток, як можна бачити на рисунку 6.4. Напис знизу показує поточного вибраного персонажа, написи зверху показують персонажа, на якого наведено курсор мишки і його кістку, на яку наведено курсор.

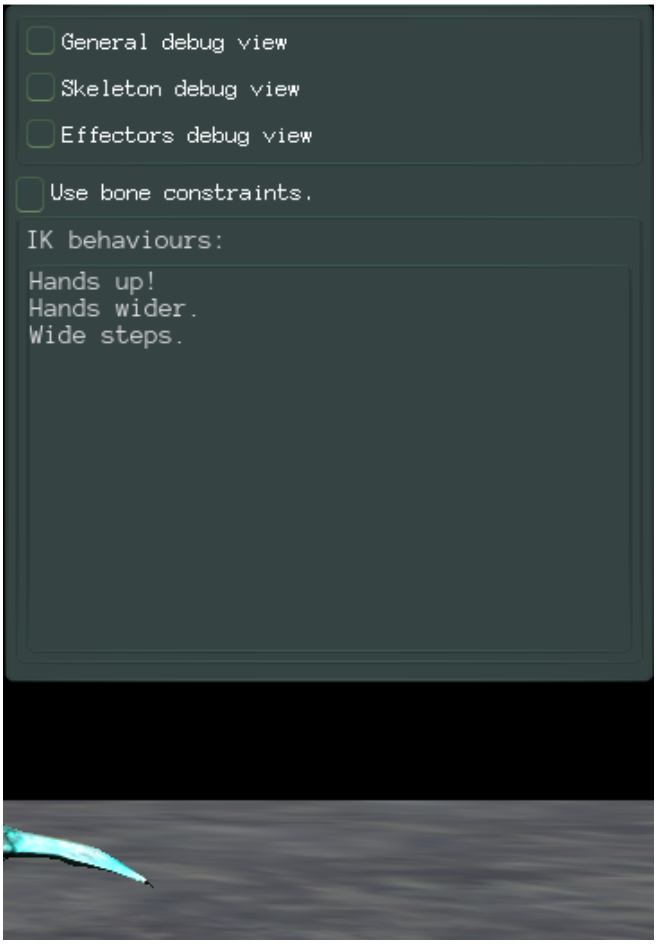

Рисунок 6.4 — Меню перемикання виду, обмежень і процедурних поведінок

На меню, що показано на рисунку 6.4 можна також бачити перемикач обмежень, цей перемикач показує і дозволяє змінювати режим застосування обмежень поворотів кісток. У програмі для персонажів створено їх обмеження для кісток ліктів, плечей, хребта, саме вони дозволяють робити рух більш природнім.

Під перемикачем режиму обмежень можна побачити список процедурних ефектів, це ефекти, які застосуються у реальному часі до персонажа, якщо він програє анімацію у проміжному форматі, їх вплив також може бути частиною експорту традиційних анімацій. У системі реалізовано 3 ефекти:

— «Руки догори» - персонаж підійме руки догори, як при погрозі пострілу, але поточна анімація все же буде на них впливати.

— «Руки вбоки» - персонаж спробує збільшити відстань між долонями рук, поточна анімація все ще буде впливати на положення рук. Ефект може бути

корисним при переносі анімацій поміж персонажами із різним співвідношенням ширини плечей і стегон.

— «Довгі кроки» - персонаж буде робити кроки довшими, як ніби він поспішає.

На рисунку 6.5 зображено ліве меню, яке дозволяє перемикатися між персонажами, вмикати програш традиційної анімації або анімації у проміжному форматі, а також експортувати поточну анімацію із проміжного формату до традиційного.

Рисунок 6.5 — Меню вибору, конвертації і експорту анімацій

Після закінчення роботи користувач може натиснути на Escape, щоб закрити програму.

ВИСНОВКИ

У ході аналізу предметної області було виявлено, що на даний момент не існує задовільного рішення проблеми уніфікації анімацій на персонажах із відмінними скелетами. Ті рішення, що існують, надзвичайно вузьконаправлені і не вирішують проблему, якщо скелети структурно відрізняються.

Розроблений програмний продукт дозволяє переносити анімації між різними персонажами і додавати до них процедурні елементи.

Проведено огляд методів і засобів розробки програмної системи. Обґрунтовано вибір створення програмної системи, заснованої на відкритому 3Д рушію із компонентною архітектурою. Це дозволило сконцентрувати увагу на вирішенні функціональних проблем, а не застрягнути на низькорівневому рівні.

Було досліджено різноманітні техніки та алгоритми комп'ютерної скелетної анімації, прямої і інверсної кінематики, було виявлено різноманітні сфери, для яких можна застосовувати дані прийоми і вибрано найбільш ефективні для реалізації програмного продукту.

Користувачами даної системи є професійні аніматори, перед якими стоїть задача розробки і художнього оформлення багатьох схожих анімацій для різних персонажів із різними скелетами. Програмне забезпечення може використовуватись на машинах, що задовольняють мінімальні вимоги до апаратної складової на базі операційної системи Windows 10.

За результатами виконання тестових завдань підтверджена коректність отриманих результатів, отже система відповідає поставленим вимогам.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Soriano. "Skeletal Animation". [Електронний ресурс] Soriano, Marc. // Bourns College of Engineering. — 2011-01-05. — Режим доступу: http://alumni.cs.ucr.edu/~sorianom/cs134_09win/lab5.htm

2. N. Magnenat-Thalmann. Joint-Dependent Local Deformations for Hand Animation and Object Grasping. [Електронний ресурс] N. Magnenat-Thalmann, R.Laperrière, D. Thalmann, Proc. Graphics Interface'88, Edmonton, 1988 — pp.26-33 Режим доступу: http://graphicsinterface.org/proceedings/gi1988/gi1988-4/

3. What is rigging? Josh Tatti [Електронний ресурс] — Режим доступу: https://conceptartempire.com/what-is-rigging/

4. Э.Фримен. Паттерны проектирования = Head First Design Patterns. / Эрик Фримен, Элизабет Фримен. — СПб: Питер, 2011. — 656 с.

5. Herbert Schildt. C++ The Complete Reference Third Edition. — Osborne McGraw-Hill, 1998.

6. Бьёрн Страуструп. Язык программирования C++ = The C++ Programming Language / Пер. с англ. — 3-е изд. — СПб.; М.: Невский диалект — Бином, 1999. — 991 с.

7. Paul, Richard . Robot manipulators: mathematics, programming, and control : the computer control of robot manipulators. MIT Press, Cambridge, Massachusetts.(1981)

8. Learn About Robots. "Robot Forward Kinematics. [Електронний ресурс] 2007-02-01. Режим доступу: https://www.learnaboutrobots.com/forwardKinematics.htm

9. Li-Chun Tommy Wang and Chih Cheng Chen. A combined optimization method for solving the inverse kinematics problems of mechanical manipulators. IEEE Transactions on Robotics and Automation, [Електронний ресурс] 7(4):489–499, 1991. Режим доступу: https://ieeexplore.ieee.org/document/86079

10. Andreas Aristidou and Joan Lasenby. FABRIK: a fast, iterative solver for the inverse kinematics problem. [Електронний ресурс] Graphical Models 73(5):243-260 · September 2011 Режим доступу:

https://www.researchgate.net/publication/220632147_FABRIK_A_fast_iterative_solver_for_the_Inverse_Kinematics_problem

11. David E. Orin and William W. Schrader. Efficient computation of the jacobian for robot manipulators. The International Journal of Robotics Research, 3(4):66–75, 1984.

12. A. Balestrino, G. De Maria, and L. Sciavicco. Robust control of robotic manipulators. In Proceedings of the 9th IFAC World Congress, volume 5, pages 2435–2440, 1984.

13. R. Fletcher. Practical methods of optimization; (2nd Ed.). Wiley-Interscience, New York, NY, USA, 1987.

14. Kwan W. Chin, B. R. von Konsky, and A. Marriott. Closed-form and generalized inverse kinematics solutions for the analysis of human motion. volume 5, pages 1911–1914, 1997.

15. Nicolas Courty and Elise Arnaud. Inverse kinematics using sequential monte carlo methods. In Proceedings of the V Conference on Articulated Motion and Deformable Objects, AMDO'08, volume 5098, pages 1–10, Mallorca, Spain, 2008. LNCS

ДОДАТОК А

Система процедурної генерації та уніфікації анімацій на основі інверсної кінематики

Специфікація

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТВ51195_27Б

Аркушів 2

Київ 2019

Позначення	Найменування	Примітки
Документація		
УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ТВ5133_18Б	Записка.docx	Текстова частина дипломної роботи
Компоненти		
УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ТВ5133_18Б 12-1	EditorScreen.h EditorScreen.cpp	Компонент основного робочого екрану.
УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ТВ5133_18Б 12-1	LoginScreen.h LoginScreen.cpp	Компонент екрану авторизації
УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ТВ5133_18Б 12-1	ProjectScreen.h ProjectScreen.cpp	Компонент екрану налаштування проекту
УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ТВ5133_18Б 12-1	IKRigAnimation.h IKRigAnimation.cpp IKAnimController.h IKAnimController.cpp IKSolver.h IKSolver.cpp	Основний компонент, що складається з трьох частин і відповідає за конвертацію і програш проміжного формату

ДОДАТОК Б

Система процедурної генерації та уніфікації анімацій на основі інверсної кінематики

Текст програми

УКР.НТУУ "КПІ". ТВ51195_17Б 12-1

Аркушів 17

2019

```
#pragma once
#include <Urho3D/Scene/LogicComponent.h>
#include <Urho3D/IK/IKEffector.h>
#include <Urho3D/IK/IKSolver.h>

#include "Data/IKRigMapping.h"
#include "Data/IKRigDef.h"
#include "Data/IKRigAnimation.h"

#include <optional>

struct IKRigAnimationState
{
    bool running_ = false;
    bool loop_ = true;
    float time_ = 0.f;
    IKRigAnimation const* anim_ = nullptr;
};

enum class Behaviour
{
    None,
    HandsUp,
    WideHands
};

class IKAnimController : public Urho3D::Component
{
    URHO3D_OBJECT(IKAnimController, Urho3D::Component);
public:
    IKAnimController(Urho3D::Context* context);
    ~IKAnimController() override = default;

    static void RegisterObject(Urho3D::Context* context);

    void Construct(IKRigDef const& def, IKRigMapping const& mapping);
    void RequestState(IKRigAnimation const* const anim, bool autoStart = true, bool loop = true);
    bool HasState() const { return curState_.has_value(); }
    bool StartState();
    bool StopState();

    float GetChainLength(Urho3D::String const& chainName) const;

       IKRigDef const& GetRigDef() const { return def_; }
       IKRigMapping const& GetMapping() const { return mapping_; }

    bool AreConstraintsEnabled() const { return solver_-
>GetFeature(Urho3D::IKSolver::CONSTRAINTS); }
    void UseConstraints(bool v) { solver_->SetFeature(Urho3D::IKSolver::CONSTRAINTS, v); }

    void SetBehaviour(Behaviour b) { if (b == currentBehaviour_) currentBehaviour_ =
Behaviour::None; else currentBehaviour_ = b; }

protected:
    void OnSceneSet(Urho3D::Scene* scene) override;

private:
    void PrepareNode();

    void CreateConstraints();

    void HandleSceneUpdate(Urho3D::StringHash eventType, Urho3D::VariantMap& eventData);
```

```
    void HandleSceneDrawableUpdateFinished(Urho3D::StringHash eventType, Urho3D::VariantMap&
eventData);
    void HandlePostRenderUpdate(Urho3D::StringHash eventType, Urho3D::VariantMap& eventData);

    Urho3D::Vector3 ApplyBehaviour(Urho3D::Vector3 const& pos, IKRigDef::Chain const& chain);

private:
    IKRigDef def_;
    IKRigMapping mapping_;
    std::optional<IKRigAnimationState> curState_;
       Urho3D::IKSolver* solver_;
       Urho3D::HashMap<Urho3D::String, Urho3D::IKEffector*> activeEffectors_;

    Behaviour currentBehaviour_;
}
#include "IKAnimController.h"
#include <Urho3D/Core/Context.h>
#include <Urho3D/Core/CoreEvents.h>
#include <Urho3D/Core/Object.h>
#include <Urho3D/Scene/Scene.h>
#include <Urho3D/Scene/SceneEvents.h>

#include <Urho3D/IK/IKConstraint.h>

#include "Common/Match.h"
#include <Urho3D/Scene/Component.h>
#include "Urho3D/Graphics/DebugRenderer.h"

using namespace Urho3D;

namespace
{
    char const* const ROOT_BONE = "Root";
}

IKAnimController::IKAnimController(Context* context)
    : Component(context) {}

void IKAnimController::RegisterObject(Context* context)
{
    context->RegisterFactory<IKAnimController>("BRIK");
}

void IKAnimController::Construct(IKRigDef const& def, IKRigMapping const& mapping)
{
    def_ = def;
    mapping_ = mapping;
    PrepareNode();
}

void IKAnimController::RequestState(IKRigAnimation const* const anim, bool autoStart /*= true*/,
bool loop /*= true*/)
{
    if (!anim)
    {
        return;
    }
    curState_.reset();
    curState_ = IKRigAnimationState{ autoStart, loop, 0.f, anim };
}

bool IKAnimController::StartState()
{
    if (curState_.has_value())
    {
```

```
        curState_->running_ = true;
        return true;
    }
    return false;
}

bool IKAnimController::StopState()
{
    if (curState_.has_value())
    {
        curState_->running_ = false;
        return true;
    }
    return false;
}

float IKAnimController::GetChainLength(Urho3D::String const & chainName) const
{
    auto chain = *std::find_if(def_.chains.Begin().ptr_, def_.chains.End().ptr_, [&](auto const&
chain) { return chain.name == chainName; });
    auto parent = GetNode()->GetChild(*mapping_.boneMapping[chain.origin], true);
    auto child = GetNode()->GetChild(*mapping_.boneMapping[chain.end], true);
    auto result = 0.f;
    while (child != parent)
    {
        auto const wpos1 = child->GetWorldPosition();
        auto const wpos2 = child->GetParent()->GetWorldPosition();
        result += (wpos1 - wpos2).Length();
        child = child->GetParent();
    }
    return result;
}

void IKAnimController::OnSceneSet(Scene* scene)
{
    if (scene && IsEnabledEffective())
    {
        SubscribeToEvent(scene, E_SCENEUPDATE, URHO3D_HANDLER(IKAnimController,
HandleSceneUpdate));
        SubscribeToEvent(scene, E_SCENEDRAWABLEUPDATEFINISHED, URHO3D_HANDLER(IKAnimController,
HandleSceneDrawableUpdateFinished));
        SubscribeToEvent(scene, E_POSTRENDERUPDATE, URHO3D_HANDLER(IKAnimController,
HandlePostRenderUpdate));
    }
    else if (!scene)
    {
        UnsubscribeFromEvent(E_SCENEUPDATE);
        UnsubscribeFromEvent(E_SCENEDRAWABLEUPDATEFINISHED);
        UnsubscribeFromEvent(E_POSTRENDERUPDATE);
    }
}

void IKAnimController::PrepareNode()
{
    for (auto const& chain : def_.chains)
    {
        /*************************************************************************/
        /*if (chain.name != "r-arm" && chain.name != "l-arm" && chain.name != "l-leg" &&
chain.name != "r-leg")
        {
            continue;
        }*/
        /*************************************************************************/

        auto endBoneNode = GetNode()->GetChild(mapping_.boneMapping[chain.end], true);
```

```
        auto newEffector = endBoneNode->CreateComponent<IKEffector>();
        auto const chainLength = reduce_to_parent(
            GetNode(),
            mapping_.boneMapping[chain.origin],
            endBoneNode->GetName(),
            [](Node*) { return 1; });
        newEffector->SetChainLength(chainLength);
        activeEffectors_[chain.name] = newEffector;
    }

    CreateConstraints();


    auto const& rootName = mapping_.boneMapping[ROOT_BONE];
    solver_ = GetNode()->GetChild(rootName, true)->CreateComponent<IKSolver>();
    solver_->SetAlgorithm(IKSolver::CCD);
    solver_->SetFeature(IKSolver::AUTO_SOLVE, false);
    solver_->SetFeature(IKSolver::CONSTRAINTS, false);
    //solver_->SetFeature(IKSolver::TARGET_ROTATIONS, true); // maybe later
}

void IKAnimController::CreateConstraints()
{
    //Mutant
    if (auto elbow = GetNode()->GetChild("Mutant:RightForeArm", true))
    {
        auto constraint = elbow->CreateComponent<IKConstraint>();
        constraint->SetConstraintType(IKConstraint::ConstraintType::Hinge);
        constraint->SetAxis(Vector3(0, -1, 0));
        constraint->SetRange({ -30.f, 175.f });
    }
    if (auto knee = GetNode()->GetChild("Mutant:RightLeg", true))
    {
        auto constraint = knee->CreateComponent<IKConstraint>();
        constraint->SetConstraintType(IKConstraint::ConstraintType::Hinge);
        constraint->SetAxis(Vector3(-1, 0, 0));
        constraint->SetRange({ 0.f, 100.f });
    }
    if (auto knee = GetNode()->GetChild("Mutant:LeftLeg", true))
    {
        auto constraint = knee->CreateComponent<IKConstraint>();
        constraint->SetConstraintType(IKConstraint::ConstraintType::Hinge);
        constraint->SetAxis(Vector3(-1, 0, 0));
        constraint->SetRange({ 0.f, 100.f });
    }

    //Kachujin
    if (auto elbow = GetNode()->GetChild("LeftForeArm", true))
    {
        auto constraint = elbow->CreateComponent<IKConstraint>();
        constraint->SetConstraintType(IKConstraint::ConstraintType::Hinge);
        constraint->SetAxis(Vector3(0, 0, 1));
        constraint->SetRange({ 0.f, 175.f });
    }
    if (auto elbow = GetNode()->GetChild("RightForeArm", true))
    {
        auto constraint = elbow->CreateComponent<IKConstraint>();
        constraint->SetConstraintType(IKConstraint::ConstraintType::Hinge);
        constraint->SetAxis(Vector3(0, 0, -1));
        constraint->SetRange({ 0.f, 175.f });
    }
    if (auto knee = GetNode()->GetChild("RightLeg", true))
    {
        auto constraint = knee->CreateComponent<IKConstraint>();
        constraint->SetConstraintType(IKConstraint::ConstraintType::Hinge);
```

```
        constraint->SetAxis(Vector3(1, 0, 0));
        constraint->SetRange({ 0.f, 100.f });
    }
    if (auto knee = GetNode()->GetChild("LeftLeg", true))
    {
        auto constraint = knee->CreateComponent<IKConstraint>();
        constraint->SetConstraintType(IKConstraint::ConstraintType::Hinge);
        constraint->SetAxis(Vector3(1, 0, 0));
        constraint->SetRange({ 0.f, 100.f });
    }
}

void IKAnimController::HandleSceneUpdate(Urho3D::StringHash eventType, Urho3D::VariantMap&
eventData)
{
}

void IKAnimController::HandleSceneDrawableUpdateFinished(Urho3D::StringHash eventType,
Urho3D::VariantMap& eventData)
{
    if (!curState_.has_value() || !curState_->running_)
    {
        return;
    }

    float const dt = eventData[Update::P_TIMESTEP].GetFloat();
    float const frameTime = IKRigAnimation::GetTimeStep();
    curState_->time_ += dt;
    if (curState_->time_ > curState_->anim_->length && !curState_->loop_)
    {
        curState_->running_ = false;
        return;
    }
    if (curState_->time_ > curState_->anim_->length)
    {
        curState_->time_ -= curState_->anim_->length;
    }

    if(false)//root transfrom
    {
        auto const nextPose = std::find_if(curState_->anim_->rootTrack.poses.Begin().ptr_,
curState_->anim_->rootTrack.poses.End().ptr_, [&](auto const& pose) { return curState_->time_ <
pose.time; });
        auto const lastPose = nextPose - 1;
        auto const lerpCoef = (curState_->time_ - lastPose->time) / frameTime;
        auto const lerped = Lerp(lastPose->effectorTransform, nextPose->effectorTransform,
lerpCoef);
        GetNode()->GetChild(mapping_.boneMapping[ROOT_BONE], true)->SetTransform(lerped);
    }

    for (auto const& trackPair : curState_->anim_->tracks)
    {
        auto const& track = trackPair.second_;
        auto relativeEffector = activeEffectors_.Find(track.chainName);
        if (relativeEffector == activeEffectors_.End())
        {
            continue;
        }

        auto const nextPose = std::find_if(track.poses.Begin().ptr_, track.poses.End().ptr_,
[&](auto const& pose) { return curState_->time_ < pose.time; });
        auto const lastPose = nextPose - 1;
        auto const lerpCoef = (curState_->time_ - lastPose->time) / frameTime;
        auto const lerpedPos = Lerp(lastPose->effectorTransform.Translation(), nextPose-
>effectorTransform.Translation(), lerpCoef);
```

```
        auto const lerpedRot = Lerp(lastPose->effectorTransform.Rotation(), nextPose-
>effectorTransform.Rotation(), lerpCoef);

        auto const myChainLength = GetChainLength(track.chainName);
        auto const chainScale = myChainLength / track.chainLength; //leave scale for now;
        auto const scaledPos = lerpedPos * chainScale;

        auto const lerpedTransform = Matrix3x4(scaledPos, lerpedRot, 1.0f);

        auto chainIt = std::find_if(def_.chains.Begin().ptr_, def_.chains.End().ptr_, [&](auto
const& chain) { return chain.name == track.chainName; });
        auto parentBone = GetNode()->GetChild(mapping_.boneMapping[chainIt->origin], true);
        auto parentWorldTransform = GetNode()->GetWorldTransform().Inverse() * parentBone-
>GetWorldTransform();
        parentWorldTransform.SetRotation(Matrix3::IDENTITY);

        auto resultingEffectorPos = (GetNode()->GetWorldTransform() * parentWorldTransform *
lerpedTransform).Translation();
        if (GetNode()->GetNameHash() != curState_->anim_->originalCharacter)
        {
            resultingEffectorPos.z_ *= -1;
        }
        resultingEffectorPos = ApplyBehaviour(resultingEffectorPos, *chainIt);
        relativeEffector->second_->SetTargetPosition(resultingEffectorPos);

        /*auto effectorBone = GetNode()->GetChild(mapping_.boneMapping[chainIt->end], true);
        effectorBone->SetRotation(lerpedRot);*/

        GetScene()->GetComponent<DebugRenderer>()->AddSphere({ resultingEffectorPos, 0.1f },
Color::RED, false);
        //relativeEffector->second_->SetTargetRotation(parentWorldTransform.Rotation() *
lerpedRot * parentWorldTransform.Rotation().Inverse());
    }
    solver_->Solve();
}

void IKAnimController::HandlePostRenderUpdate(StringHash eventType, VariantMap& eventData)
{
    if (solver_)
    {
        solver_->DrawDebugGeometry(false);
    }
}
```

Vector3 IKAnimController::ApplyBehaviour(Vector3 const & pos, IKRigDef::Chain const & chain)

{

auto result = pos;

switch (currentBehaviour_)

{

case Behaviour::None:

{

break;

```
        }
            break;
        case Behaviour::HandsUp:
        {
            if (chain.name != "l-arm" && chain.name != "r-arm")
            {
                break;
            }
            auto arm = GetNode()->GetChild(mapping_.boneMapping[chain.origin], true)->GetWorldPosition();
            auto local_mini = (pos - arm) * -0.03f;
            result = arm + Vector3::UP * 0.3f + GetNode()->GetWorldDirection() * 0.3f + local_mini;
            /*result.x_ *= 0.1;
            result.z_ *= 0.01;*/
        }
            break;
        case Behaviour::WideHands:
        {
            if (chain.name != "l-arm" && chain.name != "r-arm")
            {
                break;
            }
            auto whips = GetNode()->GetChild(mapping_.boneMapping[ROOT_BONE], true)->GetWorldPosition();
            auto wide = pos - whips;
            auto xsign = Sign(wide.x_);
            auto chainLen = GetChainLength(chain.name);
            auto midres = pos + Vector3::RIGHT * xsign * 0.25f;
```

```
            auto armNode = GetNode()->GetChild(mapping_.boneMapping[chain.origin],
true);
            auto arm = armNode->GetWorldPosition();
            auto endNode = GetNode()->GetChild(mapping_.boneMapping[chain.end],
true);
            auto tors = (midres - arm).Normalized();
            result = arm + tors * chainLen;// *(endNode->GetParent() == armNode ? 1.f :
0.8f);

        }
            break;
        default:
            break;
        }
        return result;
    }
```

```
#pragma once
#include <Urho3D/Core/StringUtils.h>
#include <Urho3d/Graphics/Animation.h>
#include <Urho3d/Graphics/AnimatedModel.h>
#include <Urho3D/Math/Vector3.h>
#include <Urho3d/Resource/JSONFile.h>
#include <variant>

#include "Common/Error.h"
#include "Data/IKRigDef.h"
#include "Data/IKRigMapping.h"

struct IKRigChainPose
{
    IKRigChainPose(float t, Urho3D::Matrix3x4 const& p)
        : time(t), effectorTransform(p) {}
    float time;
    Urho3D::Matrix3x4 effectorTransform;
};

struct IKRigChainTrack
{
    Urho3D::String chainName;
    float chainLength;
    Urho3D::Vector<IKRigChainPose> poses;
};

class IKRigAnimation
{
public:
```

```
        IKRigAnimation() = default;
    ~IKRigAnimation() = default;

    using CreationResult = std::variant<IKRigAnimation, Error>;
    static CreationResult LoadFromJson(Urho3D::JSONValue const& value);
    static CreationResult CreateFromAnimation(Urho3D::Animation* anim, Urho3D::Node*
originalModel, IKRigDef const& rigDef, IKRigMapping const& rigMapping);
    void SaveToJson(Urho3D::JSONValue& value) const;

    static constexpr float GetTimeStep() { return 0.033f; }

    Urho3D::String name;
    IKRigDef rigDef;
    float length = 0.0f;
        Urho3D::HashMap<Urho3D::String, IKRigChainTrack> tracks;
    IKRigChainTrack rootTrack;
    int trackCount = 0;

    //HACK - reverse z if needed
    Urho3D::StringHash originalCharacter;
        };
```

```
#include "IKRigAnimation.h"

#include <Urho3D/Graphics/AnimationController.h>
#include "Common/Match.h"
#include "System/IKAnimController.h"
#include <variant>

using namespace Urho3D;
namespace JsonKeys
{
    char const* const NAME = "name";
    char const* const LENGTH = "length";
    char const* const TIME = "time";
    char const* const ENDPOS = "endpos";
}

IKRigAnimation::CreationResult IKRigAnimation::LoadFromJson(JSONValue const& value)
{
    return Error{};
}

IKRigAnimation::CreationResult IKRigAnimation::CreateFromAnimation(Animation* anim, Node*
originalModel, IKRigDef const& rigDef, IKRigMapping const& rigMapping)
{
    IKRigAnimation result;
    result.name = "ik:" + rigDef.name + ":" + anim->GetName();
    result.length = anim->GetLength();
    result.rigDef = rigDef;
    result.originalCharacter = originalModel->GetNameHash();
    for (auto const& chain : rigDef.chains)
    {
        auto newTrack = IKRigChainTrack{};
        newTrack.chainName = chain.name;
        newTrack.chainLength = originalModel->GetComponent<IKAnimController>()-
>GetChainLength(chain.name);
        result.tracks[chain.name] = newTrack;
    }
    result.rootTrack.chainName = "Root";

    auto animatedModel = originalModel->GetComponent<AnimatedModel>();
    auto animState = animatedModel->AddAnimationState(anim);
    animState->SetLayer(0);
    animState->SetLooped(false);
```

```
    float const timeStep = GetTimeStep();
    for (float curTime = 0.f; curTime < result.length; curTime += timeStep)
    {
        animState->SetWeight(1.f);
        float const clampedTime = Clamp(curTime, 0.0f, animState->GetLength());
        animState->SetTime(clampedTime);
        animState->Apply();
        originalModel->MarkDirty();
        for (auto const& chain : rigDef.chains)
        {
            auto const originNode = originalModel-
>GetChild(*rigMapping.boneMapping[chain.origin], true);
            auto originWorldTransform = originalModel->GetWorldTransform().Inverse() *
originNode->GetWorldTransform();
            originWorldTransform.SetRotation(Matrix3::IDENTITY);

            auto const effectorNode = originalModel->GetChild(*rigMapping.boneMapping[chain.end],
true);
            auto const effectorWorldTransform = originalModel->GetWorldTransform().Inverse() *
effectorNode->GetWorldTransform();
            auto finalEffectorTransform =  originWorldTransform.Inverse() *
effectorWorldTransform;
            finalEffectorTransform.SetRotation(effectorNode->GetRotation().RotationMatrix());
            result.tracks[chain.name].poses.EmplaceBack(curTime, finalEffectorTransform);
        }

        auto rootBone = originalModel->GetChild(*rigMapping.boneMapping["Root"], true);
        result.rootTrack.poses.EmplaceBack(curTime, rootBone->GetTransform());
    }
    result.trackCount = result.tracks.Size();
    return result;
}
void IKSolver::SolveInternalFABRIK()
{
    int iteration = data_.maxIterations_;
    float tolerance_squared = data_.tolerance_ * data_.tolerance_;
    bool done = false;

    while (iteration-- > 0)
    {
        Vector3 root_position;
        for (auto& island : data_.chainTree_.islands_)
        {
            auto& root_chain = island.rootChain_;
            auto root_node = root_chain.nodes_.Back();
            root_position = root_node->GetWorldPosition();

            SolveInternalFABRIKForwards(root_chain);
            if (features_ & CONSTRAINTS)
            {
                SolveInternalFABRIKBackwardsWConstraints(root_chain, root_position);
            }
            else
            {
                SolveInternalFABRIKBackwards(root_chain, root_position);
            }
        }

        int done_count = 0;
        for (auto eff : effectorList_)
        {
            auto diff = eff->GetTargetPosition() - eff->GetNode()->GetWorldPosition();
            if (diff.LengthSquared() <= tolerance_squared)
            {
                done_count++;
```

```
            }
        }
        done = done_count == effectorList_.Size();
        if (done)
        {
            break;
        }
    }
}

Vector3 IKSolver::SolveInternalFABRIKForwards(Chain& chain)
{
    int average_count = 0;
    Vector3 target_position;

    for (auto& child : chain.children_)
    {
        auto child_base_pos = SolveInternalFABRIKForwards(child);
        target_position += child_base_pos;
        ++average_count;
    }

    if (average_count == 0)
    {
        auto eff = chain.nodes_.Front()->GetComponent<IKEffector>();
        target_position = eff->GetTargetPosition();
    }
    else
    {
        target_position /= (float)average_count;
    }

    for (int node_idx = 0; node_idx < (chain.nodes_.Size() - 1); ++node_idx)
    {
        auto child_node = chain.nodes_[node_idx + 0];
        auto child_ik = child_node->GetComponent<IKNode>();
        auto parent_node = chain.nodes_[node_idx + 1];
        auto parent_ik = parent_node->GetComponent<IKNode>();

        auto const curToChildWorld = child_node->GetWorldPosition() - parent_node-
>GetWorldPosition();
        child_node->SetWorldPosition(target_position);
        auto const to_target_from_parent = (target_position - parent_node-
>GetWorldPosition()).Normalized();
        parent_node->SetWorldPosition(target_position + (to_target_from_parent * -child_ik-
>segmentLength_));
        parent_ik->SetDirection(to_target_from_parent, curToChildWorld);
        parent_node->MarkDirty();
        child_node->SetPosition(child_ik->originalPos_);

        target_position = parent_node->GetWorldPosition();
    }
    chain.nodes_.Back()->MarkDirty();

    return target_position;
}

void IKSolver::SolveInternalFABRIKBackwards(Chain& chain, Vector3 const& rootPos)
{
    Vector<Vector3> prevWorldPoses; prevWorldPoses.Reserve(chain.nodes_.Size());
    for (auto node : chain.nodes_)
    {
        prevWorldPoses.Push(node->GetWorldPosition());
    }
    int node_idx = chain.nodes_.Size() - 1;
```

```
    if (node_idx > 1)
    {
        auto base = chain.nodes_[node_idx];
        base->SetWorldPosition(rootPos);
        base->MarkDirty();
    }
    Vector3 lastPos;
    while (node_idx-- > 0)
    {
        auto child_node = chain.nodes_[node_idx + 0];
        auto child_ik = child_node->GetComponent<IKNode>();
        auto parent_node = chain.nodes_[node_idx + 1];
        auto parent_ik = parent_node->GetComponent<IKNode>();

        auto const target_position = prevWorldPoses[node_idx];
        auto const last_dir = prevWorldPoses[node_idx] - prevWorldPoses[node_idx + 1];
        auto const to_target_from_parent = (target_position - parent_node-
>GetWorldPosition()).Normalized();
        parent_ik->SetDirection(to_target_from_parent, last_dir);
        parent_node->MarkDirty();
        lastPos = child_node->GetWorldPosition();
    }
    chain.nodes_.Back()->MarkDirty();
    for (auto& child : chain.children_)
    {
        SolveInternalFABRIKBackwards(child, lastPos);
    }
}

void IKSolver::SolveInternalFABRIKBackwardsWConstraints(Chain& chain, Vector3 const& rootPos)
{
    Vector<Vector3> prevWorldPoses; prevWorldPoses.Reserve(chain.nodes_.Size());
    for (auto node : chain.nodes_)
    {
        prevWorldPoses.Push(node->GetWorldPosition());
    }
    int node_idx = chain.nodes_.Size() - 1;
    if (node_idx > 1)
    {
        auto base = chain.nodes_[node_idx];
        base->SetWorldPosition(rootPos);
        base->MarkDirty();
    }
    Vector3 lastPos;
    while (node_idx-- > 0)
    {
        auto child_node = chain.nodes_[node_idx + 0];
        auto child_ik = child_node->GetComponent<IKNode>();
        auto parent_node = chain.nodes_[node_idx + 1];
        auto parent_ik = parent_node->GetComponent<IKNode>();

        auto const target_position = prevWorldPoses[node_idx];
        auto const last_dir = prevWorldPoses[node_idx] - prevWorldPoses[node_idx + 1];
        auto const to_target_from_parent = (target_position - parent_node-
>GetWorldPosition()).Normalized();
        parent_ik->SetDirection(to_target_from_parent, last_dir);
        parent_node->MarkDirty();

        if (features_ & CONSTRAINTS)
        {
            TryApplyConstraint(parent_node);
        }
        parent_node->MarkDirty();
        lastPos = child_node->GetWorldPosition();
    }
```

```
    chain.nodes_.Back()->MarkDirty();
    for (auto& child : chain.children_)
    {
        SolveInternalFABRIKBackwardsWConstraints(child, lastPos);
    }

    /*for (auto node : chain.nodes_)
    {
        TryApplyConstraint(node);
    }*/
}


void IKSolver::SolveInternalCCD()
{
    int iteration = data_.maxIterations_;
    float tolerance_squared = data_.tolerance_ * data_.tolerance_;
    bool done = false;

    while (iteration-- > 0)
    {
        for (auto& island : data_.chainTree_.islands_)
        {
            auto& root_chain = island.rootChain_;
            SolveInternalCCDChain(root_chain);

        }

        int done_count = 0;
        for (auto eff : effectorList_)
        {
            auto diff = eff->GetTargetPosition() - eff->GetNode()->GetWorldPosition();
            if (diff.LengthSquared() <= tolerance_squared)
            {
                done_count++;
            }
        }
        done = done_count == effectorList_.Size();
        if (done)
        {
            break;
        }
    }
}

void IKSolver::SolveInternalCCDChain(Chain& chain)
{
    int average_count = 0;
    Vector3 target_position;

    auto eff = chain.nodes_.Front()->GetComponent<IKEffector>();
    if (!eff)
    {
        return;
    }
    target_position = eff->GetTargetPosition();

    for (int node_idx = 0; node_idx < (chain.nodes_.Size() - 1); ++node_idx)
    {
        auto effector_node = chain.nodes_[0];
        auto cur_parent_node = chain.nodes_[node_idx + 1];
        auto cur_parent_ik = cur_parent_node->GetComponent<IKNode>();

        auto const curToEffectorWorld = effector_node->GetWorldPosition() - cur_parent_node-
>GetWorldPosition();
```

```
        auto const to_target_from_parent = (target_position - cur_parent_node-
>GetWorldPosition()).Normalized();
        cur_parent_ik->SetDirection(to_target_from_parent, curToEffectorWorld);
        cur_parent_node->MarkDirty();

        if (features_ & CONSTRAINTS)
        {
            TryApplyConstraint(cur_parent_node);
        }
    }
    chain.nodes_.Back()->MarkDirty();
    return;
}
void IKSolver::RebuildChainTrees()
{
    solverTreeValid_ = (ik_solver_rebuild_chain_trees(solver_) == 0);
    ik_calculate_rotation_weight_decays(&solver_->chain_tree); //NOT IMPLEMENTING THAT!

    chainTreesNeedUpdating_ = false;

#if MY_OLD_IMPL
    /*MY*/
    {
        data_.chainTree_.islands_.Clear();
        HashMap<StringHash, NodeMarking> involvedNodes;
        for (auto eff : effectorList_)
        {
            int chain_length_counter = eff->GetChainLength() == 0 ? -1 : (int)eff-
>GetChainLength();
            auto node = eff->GetNode();
            for (; node != nullptr; node = node->GetParent())
            {
                NodeMarking current_marking = NodeMarking::None;
                NodeMarking marking = NodeMarking::Section;
                if (chain_length_counter == 0)
                    marking = NodeMarking::Split;
                if (!involvedNodes.Contains(node->GetNameHash()))
                {
                    involvedNodes[node->GetNameHash()] = marking;
                }
                else
                {
                    if (chain_length_counter != 0)
                        current_marking = marking;
                }
                if (chain_length_counter-- == 0)
                    break;
            }
        }
        RebuildChainTreesRecursive(nullptr, data_.tree_, data_.tree_, involvedNodes);
    }
    /*MY*/
#endif

}

void IKSolver::RebuildChainTreesRecursive(Chain* chain_current, Node* node_base, Node*
node_current, HashMap<StringHash, NodeMarking>& involved_nodes)
{
    int marked_children_count;
    Node* child_node_base = node_base;
    Chain* child_chain = chain_current;

    NodeMarking marking = NodeMarking::None;
    if (involved_nodes.Contains(node_current->GetNameHash()))
```

```
    {
        marking = involved_nodes[node_current->GetNameHash()];
        involved_nodes.Erase(node_current->GetNameHash());
    }

    switch (marking)
    {
        /*
         * If this node was marked as the base of a chain then split the chain
         * at this point by moving the pointer to the base node down the tree
         * to the current node and set the current chain to NULL so a new
         * island is created (this is necessary because all children of this
         * node are necessarily part of an isolated tree).
         */
    case NodeMarking::Split:
        child_node_base = node_current;
        chain_current = nullptr;
        break;
        /*
         * If this node is not marked at all, cut off any previous chain but
         * continue (fall through) as if a section was marked. It's possible
         * that there are isolated chains somewhere further down the tree.
         */
    case NodeMarking::None:
        node_base = node_current;
        /* falling through on purpose */

    case NodeMarking::Section:
        /*
         * If the current node has at least two children marked as sections
         * or if (the current node is an effector node, but only if (the base
         * node is not equal to this node (that is, we need to avoid chains
         * that would have less than 2 nodes), then we must also split the
         * chain at this point.
         */
        marked_children_count = 0;
        PODVector<Node*> node_current_children;
        node_current->GetChildrenWithComponent<IKNode>(node_current_children);
        for (auto child : node_current_children)
        {
            auto invNodeIt = involved_nodes.Find(child->GetNameHash());
            if (invNodeIt != involved_nodes.End() && invNodeIt->second_ == NodeMarking::Section)
            {
                if (++marked_children_count == 2)
                    break;
            }
        }
        if ((marked_children_count == 2 || node_current->GetComponent<IKEffector>() != nullptr)
&& node_current != node_base)
        {
            Node* node;
            if (chain_current == nullptr)
            {
                /*
                 * If this is the first chain in the island, create and
                 * initialise it in the chain tree.
                 */
                data_.chainTree_.islands_.Push(ChainIsland{});
                child_chain = &(data_.chainTree_.islands_.Back().rootChain_);
            }
            else
            {
                /*
                 * This is not the first chain of the island, so create a
                 * new child chain in the current chain and initialise it.
```

```
                 */
                chain_current->children_.Push(Chain{});
                child_chain = &chain_current->children_.Back();
            }
            /*
             * Add pointers to all nodes that are part of this chain into
             * the chain's list, starting with the end node.
             */
            for (node = node_current; node != node_base; node = node->GetParent())
                child_chain->nodes_.Push(node);
            child_chain->nodes_.Push(node_base);
            /*
             * Update the base node to be this node so deeper chains are built back
             * to this node
             */
            child_node_base = node_current;
        }
        break;
    }
    PODVector<Node*> node_current_children;
    node_current->GetChildrenWithComponent<IKNode>(node_current_children);
    for (auto child_node : node_current_children)
    {
        RebuildChainTreesRecursive(child_chain, child_node_base, child_node, involved_nodes);
    }
}
```

ДОДАТОК В

Система процедурної генерації та уніфікації анімацій на основі інверсної кінематики

Опис програми

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ АПЕПС_ТВ51195_18Б 13-1

Аркушів 10

Київ 2019

АНОТАЦІЯ

Додаток містить опис основного компоненту системи процедурної генерації та уніфікації анімацій, поставлених в розділі 1, а саме:

— розробка проміжного формату для анімації;

— забезпечення конвертації у проміжний формат;

— забезпечення програшу анімації у проміжному форматі;

— надання доступу до набору процедурних ефектів для анімації;

— можливість запису анімації у традиційний формат.

Програмне забезпечення розроблене за допомогою мови програмування C++17 та ігрового рушію Urho3D.

ЗМІСТ

1. ЗАГАЛЬНІ ВІДОМОСТІ 76
2. ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ 77
3. ОПИС ЛОГІЧНОЇ СТРУКТУРИ 78
4. ТЕХНІЧНІ ЗАСОБИ, ЩО ВИКОРИСТОВУЮТЬСЯ 79
5. ВИКЛИК І ЗАВАНТАЖЕННЯ 80
6. ВХІДНІ ДАНІ 81
7. ВИХІДНІ ДАНІ 82

ЗАГАЛЬНІ ВІДОМОСТІ

У цьому додатку міститься опис основного компоненту системи процедурної генерації та уніфікації анімацій на основі інверсної кінематики, що виконує деякі із здач поставлених в розділі 1. У додатку Б міститься програмний код компоненту.

Система працює у вигляді самостійного додатку та для роботи потребує систему на базі операційної системи Windows 10.

Додаток розроблений за допомогою мови програмування C++17 та ігрового рушію Urho3D.

ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Додаток надає функціонал для роботи з індивідуальними обліковими записами користувачів і окремими проектами а саме:

— автентифікація користувачів;
— конфігурація проекту;
— перегляд анімацій на персонажах;
— конвертація анімацій у проміжний формат;
— перегляд анімацій у проміжному форматі;
— додавання процедурних ефектів до конвертованої анімації;
— експорт проміжного формату у вигляді традиційного формату анімацій.

ОПИС ЛОГІЧНОЇ СТРУКТУРИ

Згідно з архітектурою, додаток складається з декількох сцен із певними об’єктами, повний стан додатку є кінцевим автоматом із станами у вигляді 3 можливих сцен: аторизація, конфігурація і редагування.

На кожній сцені містяться об’єкти, які при десеріалізації та створенні запускають потоки подій для взаємодії один з одним, а також створюють компоненти для забезпечення власної поведінки.

Шар користувацького інтерфейсу має доступ до найбільш вискорівневих об’єктів на тривимірній сцені і спілкується з ними за допомогою відсилання подій-повідомлень.

ВИКОРИСТОВУВАНІ ТЕХНІЧНІ ЗАСОБИ

Для використання розробленого додатку користувач повинен мати персональний комп’ютер із установленою операційною системою Windows 10, а також установленими актуальними драйверами графічного прискорювача. Мінімальні технічні вимоги до пристрою користувача перераховані у розділі 6.

ВИКЛИК І ЗАВАНТАЖЕННЯ

Програма потребує інсталяції у вигляді розпакування архіву та, можливо, установлення компоненту операційної системи Microsoft Visual C++ Redistributable 2017.

Для роботи з програмою достатньо відкрити додаток за допомогою подвійного кліку по іконці додатку.

ВХІДНІ ДАНІ

Список вхідної інформації:

- опис виду інверсного скелету(з додатком постачається опис людиноподібного інверсного скелету);
- опис персонажів, їх моделей;
- описи співвідношення віртуальних скелетів персонажів із інверсним скелетом;
- список анімацій.

Уся ця інформація є частиною конфігурації проекту.

ВИХІДНІ ДАНІ

Вихідна інформація додатку:

— записані файли анімації у проміжному форматі із процедурними ефектами;

— експортована анімація у традиційному форматі.